MASTER EN SCIENCES INFORMATIQUES

Création de librairies Prolog pour le Web

Lefèvre, Julien

Award date:
2004

Namur
Institut d'informatique
Année académique 2003-2004

Création de librairies Prolog pour le Web

Mémoire de Julien Lefèvre
Licence en Informatique

Promoteur : Monsieur J-M Jacquet

Abstract: The purpose of this thesis is to present the various techniques necessary for the realization of a Prolog library making it possible to interface the world of the artificial intelligence with that of the Internet. On the one hand, the Prolog language is used to define knowledge and rules making it possible to induce news from those. This language is run by an abstract machine WAM. On the other hand, CGI interface is a link between a Web server and an external application. Lastly, the Pillow bookshop and the LogiMoo application are two case studies of an interaction between Prolog and Internet.

Résumé : Ce mémoire a pour but de présenter les différentes techniques inhérentes à la réalisation d'une librairie Prolog permettant d'interfacer le monde de l'intelligence artificielle avec celui de l'internet. D'une part le langage Prolog permet de définir des connaissances et des règles permettant d'induire de nouvelles connaissances à partir des connaissances. Ce langage est alors mis en œuvre par une machine abstraite WAM. D'autre part, l'interface CGI est un lien entre un serveur web et une application externe. Enfin, la librairie Pillow et l'application LogiMoo sont deux études de cas d'une interaction entre Prolog et Internet.

Je tiens tout d'abord à remercier mes proches pour m'avoir soutenu dans ce travail.

Ensuite, mon promoteur monsieur Jacquet, pour le suivi apporté à la réalisation de ce projet.

Table des matières

Table des figures

Table des exemples

Introduction

Prolog est reconnu comme un langage permettant de construire facilement des systèmes experts. Ces systèmes, basés sur l'intelligence artificielle, permettent de fournir des conseils sur base d'inférences posées sur les connaissances dont ils disposent.

D'autre part, la consultation moderne de documents s'effectue par le biais de la « technologie web » s'appuyant essentiellement sur les normes HTML, XML et HTTP. Internet a vu son utilisation exploser ces dernières années. En effet, le succès d'Internet n'est plus à prouver, tout comme son extension. L'augmentation de sa popularité en a fait un outil de marketing où la personnalisation du site visité est un atout commercial fort prisé ces derniers temps. La dernière évolution quant à la consultation de sites sur Internet est de fournir des conseils. Ainsi, certains cabinets d'avocats n'hésitent-ils pas à procurer des avis personnalisés sur le web.

Ces deux mondes, web et systèmes experts, doivent donc être liés afin de mixer le grand canal de communication que représente le net et la puissance de raisonnement apportée par l'intelligence artificielle. Cette liaison peut se mettre en oeuvre par le biais d'une librairie de prédicats « orientée web ».

L'objectif de ce mémoire est d'expliquer les différentes techniques mises en œuvre lors de la réalisation d'une telle librairie.

Pour pouvoir interfacer le serveur Prolog avec un client web, deux types de techniques existent : soit augmenter le langage Prolog pour qu'il prenne en compte le CGI et donc lui-même être appelé par le client web ; soit implémenter une librairie qui interfacerait les deux mondes (Prolog et Web).

Dans un premier temps, les fondements techniques utiles à la compréhension de ce mémoire seront exposés. Tout d'abord la présentation du langage Prolog, un langage déclaratif qui permet de décrire des connaissances et le but à atteindre du programme. Les programmes réalisés à l'aide de ce langage devant être exécutés, il nous faudra alors présenter l'interpréteur de ce langage, qui consiste en une machine abstraite connue sous le nom de la WAM. Ensuite, concernant maintenant l'aspect Internet la technologie CGI apporte une interface entre un serveur web et des applications externes. Enfin les tableaux partagés tels qu'utilise Linda, constituent un outil de partage et de coordination entre différentes applications.

D'autre part, dans la seconde partie traitera des applications que peut apporter l'intelligence artificielle. On y retrouvera le traitement du langage naturel, qui vise à simplifier l'interface utilisateur pour les nouvelles applications à fin les rendre plus conviviales. Il sera question également des agents intelligents et de leur agencement au sein de systèmes multi-agents et du principe de la mobilité pour ces agents. Enfin cette seconde partie se clôturera par la présentation de deux applications existantes et leur intérêt pour la création de cette librairie d'interfaçage. Ainsi nous parlerons de la librairie Pillow intégrée au Ciao Prolog et qui permet de réaliser des applications recevant des données depuis un formulaire Internet et de fournir des résultats pouvant être affichés par un navigateur web. La seconde, LogiMoo, consiste en la gestion d'un monde virtuel destiné à être manipulée via un navigateur web.

Enfin, nous conclurons en présentant les avantages et inconvénients d'une telle librairie, si elle était largement diffusée et non plus réservées à un public averti.

Partie I : Fondements

Cette première partie va d'abord traiter des bases nécessaires à la compréhension de la suite de ce mémoire, en présentant la programmation logique. On y trouve la définition du langage Prolog, de son interpréteur. Et d'un « dérivé » du Prolog sous forme binaire, le BinProlog.
Ensuite, je présenterai l'interface standard du monde Internet : le CGI. Enfin, une introduction à la technique de coordination par tableau partagé, illustrée par le langage Linda, clôturera cette première partie.

Chapitre 1 : Programmation logique

Le sujet qui nous préoccupe est d'interfacer Prolog avec le monde de l'Internet, nous allons ici nous attarder quelque peu pour fixer les bases de la programmation logique et du langage Prolog en particulier.

1. Prolog

1.1 Introduction

Tout d'abord Prolog signifie PROgrammer en LOGique. La programmation logique est du type déclaratif, dans le sens où elle décrit ce qui doit être fait, et non comment le faire. Cette technique permet de décrire sans ambiguïtés des connaissances, des hypothèses et des faits à établir. Elle constitue donc un formalisme solide décrivant le but d'un programme.

Prolog a été conçu au début des années 70 par Alain Colmerauer et ses collègues de l'université de Marseille, sur base du concept développé par R. Kowalski. Avec le Prolog, il existe également une interprétation du langage, mais celle-ci reste réservée aux initiés de ce nouveau domaine. Il aura fallu attendre la fin des années 1970 pour que D. Warren et ses collègues de l'université d'Edinburgh implémentent un compilateur Prolog. Ce dernier, la WAM, apporte une certaine stabilité et simplicité de par son architecture détaillée, ce qui a permit de rendre ce langage plus efficace et de lui permettre d'être utilisé dans des applications à caractère commercial. Depuis, de nombreuses versions commerciales de Prolog sont apparues sur le marché, chacune apportant son lot d'extensions au langage sous forme de librairies et de nouvelles fonctionnalités offertes. Certaines permettent de gérer l'interface graphique ou encore l'accès à des bases de données. Mieux encore, il est possible d'encapsuler des prédicats Prolog dans des langages orientés objet.

1.2 Syntaxe de base

La programmation logique est basée sur un sous-ensemble de la logique des prédicats, la logique du premier ordre. Une base de connaissances Prolog contient des clauses. Une clause peut être un fait ou une règle, qui décrit comment déduire de nouveaux faits par induction. Une règle se compose d'une entête et d'un corps séparés par le symbole « :- ».

On appelle également prédicats l'apparition dans une règle de l'appel à une autre règle devant être vérifiée pour que la première règle soit vérifiée. Le nom du prédicat est appelé foncteur et le nombre de ses arguments, son arité. Un prédicat est identifié de manière unique grâce à la

combinaison de son foncteur et de son arité. En d'autres termes, deux prédicats de même nom mais avec une arité différente sont jugés différents.

Voici la notation d'un prédicat : pere /2
Et la syntaxe qui lui est associée : pere (paul, pierre).

Les arguments de prédicats sont des termes qui peuvent être :

- Un entier ou un réel
- Un atome (identificateur)
- Une variable
- Une structure de données
    - A l'aide de symboles de fonctions, comme dans
      Anniversaire (paul, date(1970)).
    - Soit à l'aide de listes, comme dans
      Liste ([element1, element2, element3]).

Voici maintenant un exemple concret permettant de tirer des connaissances concernant les liens de parenté au sein d'une famille.

Exemple 1 – Règles Prolog pour définir de nouvelles connaissances

- Des faits :

```
homme(julien).
pere(marc, julien).
pere(jean, marc).
```

- Une règle :

```
Grand pere (X,Y) :- pere(X,Z), pere (Z,Y).
```

Dans l'exemple 1, les faits décrivent que Julien est un homme, que Marc est le père de Julien et que Jean est le père de Marc. Ensuite la règle grandpere permet de créer comme connaissance supplémentaire que Jean est le grand-père de Julien. Il est à noter que les prénoms en minuscules sont appelés « atomes », ils désignent des symboles et sont donc des données. Ils ne peuvent en aucun cas commencer par une majuscule, ce qui en ferait alors des variables et non plus des symboles. Ensuite, les identificateurs en majuscule sont des variables et peuvent donc prendre diverses valeurs.

La règle grandpere s'exprime de façon logique « pour tout X et Y, s'il existe un Z tel que X soit le père de Z et que ce dernier (Z) soit le père de Y, alors X est le grand-père de Y ». L'interpréteur Prolog qui va devoir résoudre ces règles par déduction, va tenter de donner des valeurs aux variables X, Y et Z à partir des faits pour que la règle soit vraie. Cette technique s'appelle l'unification (voir section suivante).

Enfin, la dernière notion de syntaxe concerne les procédures. Il s'agit d'un ensemble de clauses qui ont le même nom (foncteur). Voici l'exemple 2, la procédure grandpere qui permet de trouver les grands-pères, soit paternels, soit maternels :

Exemple 2 - Procédure Prolog

grandpere (X,Y) :- pere (X,Z), pere (Z,Y).
grandpere (X,Y) :- pere (X,Z), mere (Z,Y).

Un programme Prolog est donc un ensemble de relations (faits et règles) et une requête vérifiant qu'une relation existe dans la base de connaissance, où qu'elle peut en être déduite.

Voici l'exemple 3, un programme Prolog complet, utilisant des faits, des relations et des règles :

Exemple 3 - Programme Prolog complet

Posons les faits suivant pour représenter la relation genre entre ces deux ensembles

soit X1={luc, julie, jean} et X2={homme, femme}.
La relation genre s'exprime avec trois faits:
genre(luc, homme).
genre(julie, femme).
genre(jean, homme).

À partir de ces faits posons les règles suivantes :
Male(X) :- genre (X, homme).
Hermaphrodite(X) :- genre(X, homme),genre(X, femme).

La traduction logique est : pour tout X tel que si la relation genre (X, homme) existe et que la relation genre(X, femme) existe, alors la relation hermaphrodite (X) existe aussi.

1.3 Bases de Prolog

A) Opérateurs d'égalité et de comparaison

Le Prolog standard, contient des prédicats prédéfinis qui permettent d'unifier, de tester l'égalité ou de comparer des termes (nombres, atomes, structures).

Il est à noter que nous utiliserons ici la spécification pour le Prolog LPA, et la syntaxe peut donc varier en fonction de l'implémentation du Prolog utilisé (GNU Prolog, Bin Prolog, ...).

= Unifie deux termes
== Vérifie si deux termes sont égaux
@< Vérifie si un terme est plus petit qu'un autre
@=< Vérifie si un terme est plus petit ou égal à un autre
@> Vérifie si un terme est plus grand qu'un autre
@>= Vérifie si un terme est plus grand ou égal à un autre
\= Contrôle la non-unification de 2 termes
\== Vérifie si 2 termes ne sont pas identiques

Voici un exemple illustrant ces instructions

Papier == crayon est faux
Abracadabrant @< zygomatique est vrai

Il existe également dans Prolog des prédicats arithmétiques prédéfinis :
< Plus petit
=< Plus petit ou égal
> Plus grand
>= Plus grand ou égal
= := Égal
=\= Différent
is Évaluation d'une expression et assignation à une variable

Pour évaluer une expression avec le prédicat is, on dispose entre autres des fonctions arithmétiques suivantes :
+ Addition
- Soustraction
* Multiplication
/ Division

Par exemple

2>3+4 est faux
*6 = := v3*2* est vrai
X is 12/3 renvoie X = 4

B) Récursion et structures de données

La récursion est une technique très utilisée et efficace pour décrire des structures qui contiennent des structures semblables à elles même. Elle est également utilisée pour décrire des règles qui s'appellent elles-mêmes.

Voici un exemple de procédure récursive en Prolog. Cette procédure écrit simplement plusieurs fois un message à l'écran.

Exemple 4 - Récursivité sans fin

```
boucle :- write('Entrez un message'),
          nl,
          read(Mot),write(Mot),
          boucle.
```

Avant toute chose, il faut rappeler que boucle et récursion sont deux concepts distincts. Si même la récursivité permet de réaliser plusieurs fois la même opération et peut parfois ne jamais s'arrêter, comme le ferait une boucle, elle met en œuvre des mécanismes plus poussés, comme des allocations de ressources et la gestion du retour à la procédure appelante.

Il est évident au premier regard de l'exemple 4 que la récursion ne va jamais s'arrêter. En effet il n'existe aucune condition d'arrêt, la procédure boucle se rappelle toujours elle-même dans tous les cas. Ajouter une condition d'arrêt est donc nécessaire et est une technique fréquemment utilisée pour terminer une récursion. La condition d'arrêt est également appelée le « cas de base ».

Exemple 5 - Récursivité avec cas de base

```
boucle (fin).
boucle (_) :-    write('Tapez fin pour terminer'),
                 nl,
                 read(Mot),
                 boucle(Mot).
```

Sur cet exemple 5, on crée une procédure boucle dont la première règle constitue le cas de base. Celle-ci ne fait rien (condition vide), lorsqu'elle reçoit le terme fin, la récursivité est arrêtée. La deuxième règle de la procédure constitue la récursion et est appelée lorsque le terme passé en paramètre n'est pas « fin ». Comme le terme passé n'a aucune importance pour cette règle, on utilise la variable anonyme.
Une dernière précision : le moteur d'inférence de Prolog évalue les règles de manière séquentielle, il faut donc veiller à ce que le cas de base soit placé un premier lieu. Sinon la règle générale sera toujours appelée et le problème de la boucle infinie ne sera pas levé.

C) Les listes

Avec la récursion, la structure de liste est couramment utilisée en programmation logique. Une liste est une séquence d'éléments qui peut avoir une longueur quelconque et dont l'ordre des éléments est important. Ces éléments sont des termes, donc peuvent être à nouveau une liste.

L'exemple 6 montre plusieurs listes possibles en Prolog.

Exemple 6 - Listes en Prolog

```
[1,2,3,4,5,45].
[ pêche, prune, banane].
[X,Y,[1,2],chien].
```

Pour manipuler les listes, on utilise le symbole | qui permet de séparer le premier élément de la liste (tête) du reste de celle-ci (corps). La récursion est utilisée pour parcourir et traiter les listes.

Voici un exemple de procédure récursive manipulant les listes.

Exemple 7 - Manipulation récursive des listes

```
ecrire_liste ([] ).
ecrire_liste ( [Tete | Suite] ) :- write (Tete), nl, ecrire_liste(Suite).
```

Ici, la liste est parcourue séquentiellement pour imprimer son contenu. Le cas de base comprenant la liste vide terminera le processus d'impression.

1.4 Unification

A partir de ces faits et de ces règles, la technique d'unification est utilisée pour vérifier les requêtes. Il s'agit tout simplement d'associer une valeur (terme) à une variable et d'utiliser cette valeur dans la suite du raisonnement (déroulement du programme Prolog). Cette association s'effectue à l'aide de substitutions. Il s'agit d'un ensemble de couples « variable, terme ». Une unification peut être défaite et peut associer plusieurs valeurs à une même variable.
Deux termes sont unifiables si on peut trouver des valeurs de variables qui les rendent égaux.
L'exemple 8 propose des unifications et le résultat de l'unification entre les variables et valeurs.

Exemple 8 - Unification

Prenons ces termes :

Y = julie → la variable Y prend la valeur julie
enfant (alice, X) = enfant (alice, jean) → la variable X prend la valeur jean
enfant (alice, paul) = enfant (alice, jean) → pas d'unification possible

L'unification recourt à une substitution particulière, le MGU (most general unification), il s'agit de la substitution permettant l'unification la plus générale entre termes. La présentation de cette notion de MGU, se base sur le cours [wam5].

Unificateur le plus général

Deux termes A et B sont unifiables, s'il existe une substitution ~, telle que **A~ = B~**. Une telle substitution ~ est un unificateur de A et B.
De manière plus intuitive, un unificateur ~ de A et B instancie les variables de A et B de façon à les rendre identiques.

Si A=point(X,Y,12) et B=point(U,13,V), alors ~={X=U,Y=13,V=12} est un unificateur de A et B, puisque A~ = B~=point(U,13,12).

De même ~2={X=15,Y=13,V=12} est un autre unificateur de A et B, puisque A~2=B~2=point(15,13,12).
Lorsque deux termes sont unifiables, ils admettent plusieurs unificateurs.

Il est normal de s'intéresser aux unificateurs qui donnent pour instance commune "l'expression la plus générale" possible.
Dans l'exemple ci-dessus, l'unificateur ~ est préférable à ~2, le terme point(U,13,12) étant plus général que point(15,13,12).

Un terme A est une instance du terme B, s'il existe une substitution ~ telle que A=B~.
A est plus général que B, si B est une instance de A.
A et B sont des variantes, si A est une instance de B et B est une instance de A.
Les variantes ne diffèrent pas que par le nom des variables utilisées.

~ est un unificateur le plus général (m.g.u. – most general unifier) de deux termes A et B, si :
a) ~ est un unificateur de A et B
b) pour tout unificateur ~~de A et B A~ est plus général que A~~.

1.5 Preuves

Le but d'un programme Prolog est de prouver que la requête soumise est une relation qui existe dans les faits, les relations et les règles établies. Voici l'algorithme récursif qui est à la base de la vérification :

Prouver (requête)
1. *Pour chaque fait et tête de règle concordant à la requête :*
   Si c'est un fait : on a trouvé une preuve
   Si c'est une tête de règle :
      Pour chaque condition de la règle : prouver(condition)
      Si succès pour chaque condition : preuve existe
2. *Si aucune concordance : Echec (backtracking)*

Lors de la preuve, les faits et règles sont analysés dans l'ordre de leur apparition dans le code du programme.
Si une sous preuve échoue, la recherche revient sur ses pas pour tenter la prochaine alternative de preuve.
C'est le **backtracking**. Pour le réaliser, il faut pouvoir garder une trace des différentes alternatives et défaire les unifications de variables effectuées pendant la preuve qui a échoué. Ce moyen est expliqué dans la partie concernant l'architecture de la Machine Abstraite de Warren.
Enfin, l'endroit où l'on peut choisir la règle à traiter lorsqu'il y en a plusieurs qui correspondent est appelé un **point de choix**.

Lorsqu'une preuve est complète, elle correspond à un arbre de preuve, découpé en sous preuves. Voici un exemple qui montre un graphe de filiation.

Exemple 9 - Preuve de Programme

D'abord, les faits et les règles :

```
enfant(thomas, marc).
enfant(thomas, kevin).
enfant(kevin, antoine).
descendant (P,P).
descendant(X,Y) :- enfant(X,I), descendant(I,Y).
```

Il faut lire « l'enfant de thomas est marc» et « le descendant de X est Y».

Lorsque l'on donne la requête descendant (thomas, antoine). Voici l'arbre qui se construit.
Première étape.

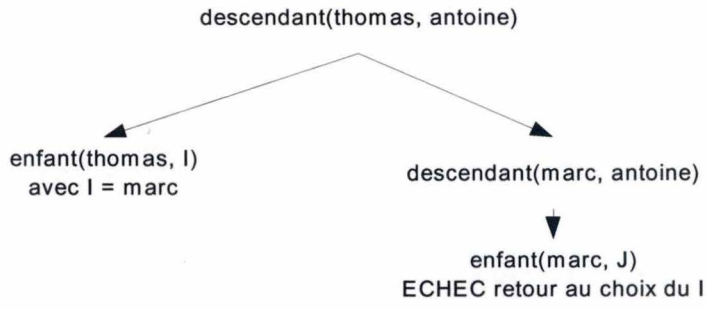

Figure 1 - arbre de résolution n°1

Là, on retourne au choix du I c'est à dire au choix du premier enfant on s'est trompé avec marc, essayons avec kevin.

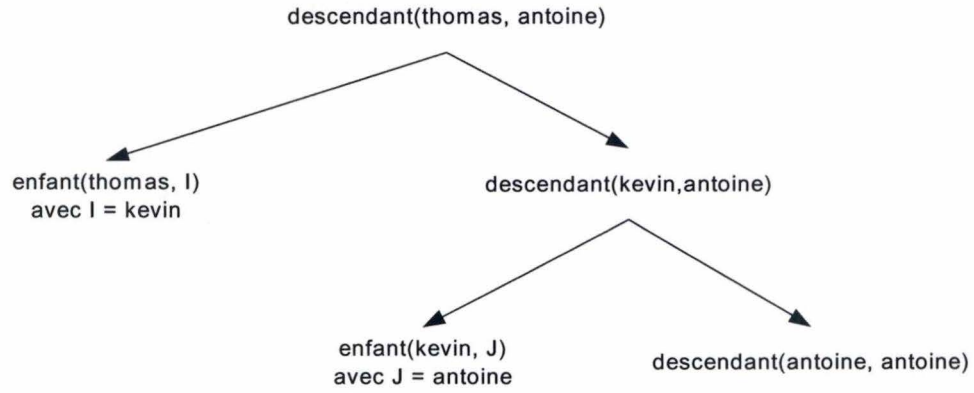

Figure 2 - arbre de résolution n°2

Ici il y a réussite avec I = kevin et J = antoine.
La preuve de ce programme Prolog pour cette requête existe et l'exécution renvoi simplement « Oui » signifiant que le fait demandé en requête fait bien partie de la connaissance du programme.

1.6 Modification du déroulement : le cut (!)

Prolog offre un prédicat système appelé **cut** qui permet de changer le déroulement procédural des programmes. La fonction principale du cut est de réduire l'espace de recherche de solutions. Il est utilisé principalement :

- Lorsque l'on veut faire échouer un but sans que Prolog cherche des solutions alternatives ; le cut est utilisé en conjonction avec une autre instruction : le fail.
  Ainsi, il est possible d'implémenter la négation à l'aide ces deux opérations :
  ```
  not(X) :- X, !, fail.
  not(_).
  ```
- Lorsque l'on sait que Prolog a trouvé la bonne règle à appliquer et qu'on ne veut pas qu'il en essaye d'autres
- Lorsque l'on veut terminer la génération de solutions parce que l'on sait que Prolog a trouvé la bonne solution

L'exemple suivant illustre l'utilisation du cut

Exemple 10 - utilisation du cut

```
find(E, tree(E,_,_)):- !
find(E, tree(X,L,_)) :- E<X,! find(E,L).
find(E, tree(_,_,R)) :- find(E,R).
```

Il s'agit d'une procédure qui recherche une valeur dans un arbre binaire ordonné, c'est-à-dire qu'à partir d'un nœud et de valeur V, les valeurs inférieures à E se trouve dans le sous arbre gauche de ce nœud, et les valeurs supérieures dans celui de droite. La première clause est la clause d'arrêt lorsque la valeur E est présente dans le nœud de l'arbre. Une fois cette clause utilisée, il ne sert plus à rien de continuer les recherches. L'utilisation du cut permet l'arrêt des recherches. La deuxième clause aiguille la recherche vers le sous arbre gauche si la valeur est recherchée est inférieure à celle du nœud actuel. Là aussi il est inutile de revenir en arrière, la valeur recherchée ne pourra pas être ailleurs que dans le sous arbre gauche car l'arbre est trié.

Il existe d'autres prédicats qui permettent de contrôler le backtracking :

- *fail* : ce prédicat échoue toujours et force donc le backtracking
- *true* : ce prédicat réussit une seule fois
- *repeat* : ce prédicat réussit toujours
- *write, read, nl* : ces prédicats permettent respectivement d'écrire, de lire et d'effectuer un retour chariot ; ils réussissent une seule fois et échouent s'ils sont appelés à nouveau pendant le backtracking.

Pour conclure, ce point consacré au langage Prolog, voici quelques exemples illustrant des cas souvent traités dans le cadre de l'intelligence artificielle.

L'exemple 11 implémente un cas typique de la programmation logique, la résolution du problème des tours de Hanoi. Pour rappel, il s'agit de trois tours en bois sur lesquelles se trouvent des disques. Les X disques se trouvent empilés sur la première tour et doivent se trouver empilés sur la troisième tour. On peut déplacer un disque sur n'importe quelle tour à condition que ce soit sur une tour vide ou sur un disque de plus grande taille. Cet exemple à été implémenté et exécuté sur le système Ciao Prolog, qui sera abordé plus loin au chapitre 6.1 lors de l'étude de la librairie Pillow.

Exemple 11 - Les tours de Hanoi

```
:- module(hanoi,[main/1]).

main(_) :- write('Entrez le nombre de disques'),nl,
        read(NB), hanoi(NB).

hanoi(N) :- dohanoi(N, 1, 3, 2).

dohanoi(0, _ , _ , _ ):-!.
dohanoi(N, A, B, C)      :-
        N_1 is N-1,
```

```
        dohanoi(N_1, A, C, B),
        moveit(A, B), dohanoi(N_1, C, B, A).

moveit(F, T) :- write([move, F, -->, T]), nl.
```

2. WAM

Pour implémenter le langage Prolog, l'interpréteur le plus utilisé et le plus répandu est la Machine Abstraite de Warren. Je vais en présenter maintenant le fonctionnement en expliquant l'architecture de mémoire et les instructions.

2.1 Architecture

Afin de permettre au lecteur de visualiser plus aisément les différentes structures utilisées par la Wam, je préfère donner ce schéma récapitulatif ainsi qu'un bref descriptif, avant de présenter les points précis de l'interpréteur. Voici donc l'architecture mémoire utilisée par la machine de Warren.

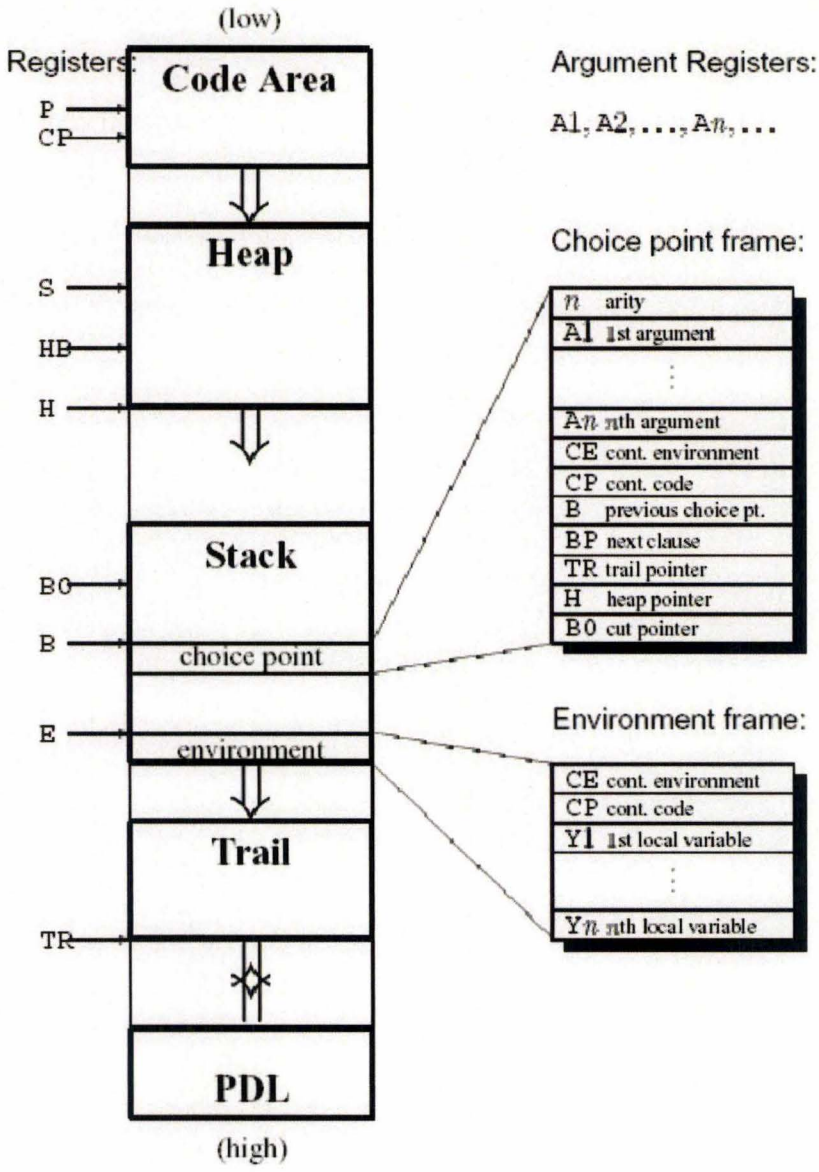

Figure 3 - architecture mémoire de la WAM

Tout d'abord il existe un tableau Code qui contient le code Prolog compilé qu'il va falloir interpréter. Il utilise deux registres : **P** qui contient toujours la prochaine instruction et **CP** qui est utilisé pour retenir l'adresse de l'instruction suivante lors du retour d'un appel de prédicat.

Le second tableau est le HEAP. Cette structure permet stocker les données : des relations et des variables

Ensuite le Stack reprend les environnements successifs de l'exécution correspondant aux appels de procédures.

On trouve encore le « Trail » qui est un tableau servant pour le mécanisme de backtracking. Il contient les adresses (dans STACK et dans HEAP) des variables qu'il faudra délier lors d'un backtracking. On n'y retrouve que les liaisons conditionnelles, celles qui pourraient donner lieu à un backtracking..

Enfin, le PDL (Push-down list) est une structure dynamique qui est un tableau d'adresses de données qui est utilisé pour la méthode d'unification.

Je vais maintenant entrer plus en détail dans l'architecture de cette machine en présentant différents aspects de son fonctionnement. À travers les points suivants je vais introduire progressivement les structures de données utilisées, à fin de bien comprendre l'utilité et le fonctionnement de chacune.

2.2 Représentation des termes et unification

La représentation interne des termes dans la WAM utilise un bloc de cellules adressables appelé Heap. Les termes sont représentés par des cellules mémoire étiquetées, c'est à dire des paires (étiquette, valeur). Une cellule prend habituellement un ou deux mots machine. L'étiquette donne le type de la représentation c'est à dire indique comment interpréter la valeur de la cellule. Il y a trois types de base : les références (étiquette REF), les foncteurs (étiquette FUN) et les termes composés (étiquette STR). Le registre **H** désigne le sommet de ce tableau.

Ces étiquettes sont utilisées comme suit pour représenter un terme en fonction de sa nature :

- une auto-référence <REF, adresse> : si le terme est une variable libre, la valeur d'une auto-référence est l'adresse de la cellule.
- une cellule de foncteur <FUN, identificateur unique du foncteur> : si le terme est une constante.
- une cellule de terme composée <STR, adresse> : si le terme est un terme composé, la valeur de la cellule donne l'adresse, sur la pile, d'une structure de données dans laquelle sont rangées dans des cellules consécutives les représentations du foncteur et des arguments du terme composé.

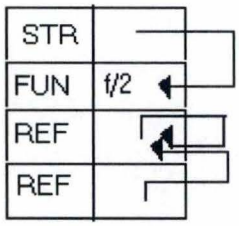

Figure 4 - représentation de Y = f(X,X) dans HEAP

À coté du tableau Heap, il existe les registres d'arguments qui permettent de garder en mémoire les arguments du terme qui est en cours de traitement, et donc l'appel de fonctions (clauses) avec passage de paramètres. Ces registres offrent également la possibilité de gérer la structure en arbre des termes et également la possibilité qu'une variable soit présente plusieurs fois dans le même terme. La WAM garde ainsi une copie des variables utilisées dans ces registres qui sont constitués suivant le même format qu'une cellule du tableau HEAP. Ces registres sont notés A[1] ... A[n].

L'unification, mécanisme qui a pour but de permettre de prouver l'appartenance d'une requête à la base de connaissances, rend deux termes égaux par substitution. Une substitution est un ensemble $\{v_1/t_1,\ldots, v_n/t_n\}$ où les v_i $(1 \leq i \leq n)$ sont n variables distinctes et chaque t_i (terme) est distinct de v_i. La variable v_i, initialement libre, ou non liée, est dite liée à t_i. On dit aussi que t_i est la valeur de liaison de v_i. Si t_i n'est pas une variable, v_i est dite instanciée.
Ces substitutions sont gardées en mémoire dans un tableau nommé PDL (pour push down list).

L'unification lie donc des variables libres (les v_i) à des termes (les t_i).

2.3 Gestion des environnements et des points de choix

Avant toute chose, il convient de formaliser un peu plus la structure de l'arbre de recherche utilisé par Prolog. Cet arbre de recherche est composé d'un ensemble d'état de recherche qui sont des triplets $< r, \sigma, b >$ où :

- r est un numéro de clause,
- σ est une substitution (sous la forme $\{ X_1 \leftarrow t_1, \ldots, X_n \leftarrow t_n\}$),
- b est une suite d'atomes B_1, B_p.

L'arbre de recherche pour le but Q_1, Q_m est un arbre où chaque nœud est occurrence d'un état de recherche. La racine est le triplet $< 0, \{\}, Q_1, \ldots, Q_m>$. Un nœud succès est un nœud de la forme $< r, \sigma, b >$ avec b vide.
Pour tout les autres nœuds, que je noterai $v : < r, \sigma , b >$ et tout les fils de v, notés $v' : < r', \sigma ', b' >$ alors :

- v et v' sont liés par le fait que B_1 (premier atome de b) est unifiable avec la tête de la clause de numéro r'.
- v est une feuille échec, s'il n'y a aucun r'. Autrement dit, s'il n'existe aucune clause qui puisse être unifiée pour continuer la preuve.

La Figure 5 montre un exemple d'arbre de recherche standard. Le prédicat p(X,Y,Z) doit être lu sous la forme X + Y = Z. Il est à noter que dans cet exemple fourni par `[wam4]` un entier est codé par le terme $s^n(0)$. Il s'agit de la fonction de successeur définie comme suit : s(X) = X+1. Elle permet de tester si un nombre est zéro ou non, sans devoir utiliser les opérateurs de comparaison. Le but recherché par l'arbre est p(s(0),0,R), p(A,B,R) qui doit être lu comme « si R est le résultat de 1+0, quelles sont les valeurs A et B telles que A+B=R ? ».

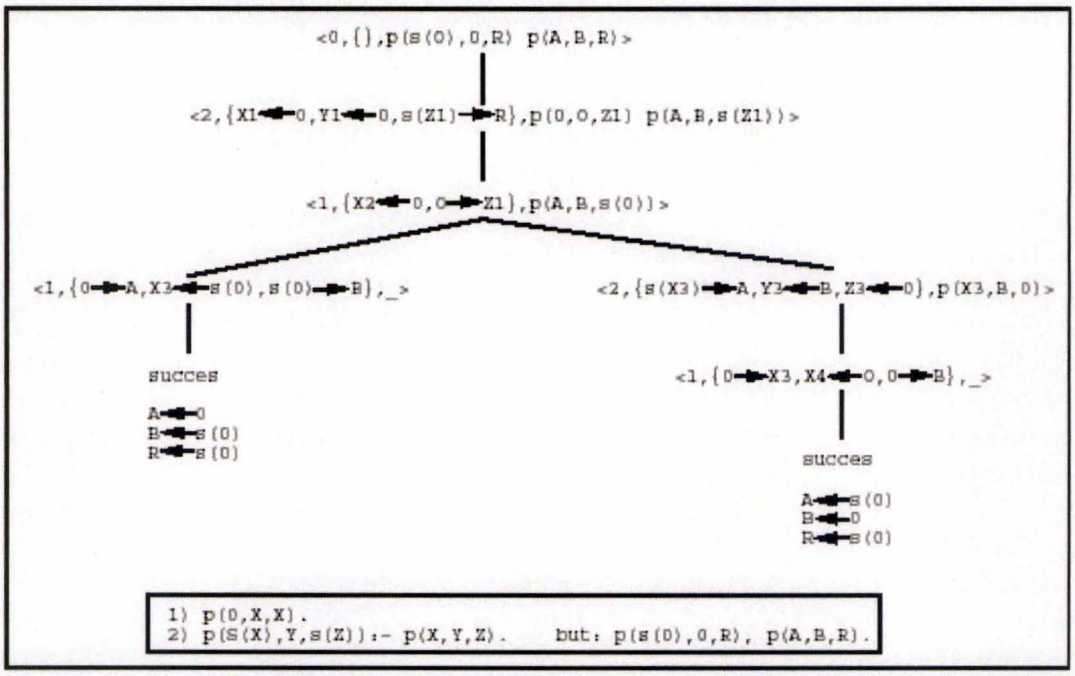

Figure 5 - exemple d'arbre de recherche standard [wam4]

La structure de l'arbre de recherche est bien entendu récursive : une occurrence de $<r, \sigma, v>$ forme un nouvel arbre de recherche standard. La stratégie de résolution standard (utilisée par Prolog) consiste en un parcours en profondeur d'abord de la gauche vers la droite. Cette stratégie peut être aisément automatisable à l'aide d'une pile appelée *pile de contrôle* ou *pile locale* (*stack*).

Ainsi, pour naviguer dans l'arbre de preuves, la WAM dispose d'une structure appelée la pile locale (stack) qui gère le contrôle de Prolog. Ce contrôle se compose de 2 phases :

- L'avancée : qui consiste en l'appel imbriqué de procédures. La WAM utilise un *environnement* où sont stockées les variables de la clause et les informations de contrôle utiles lors de la sortie du bloc courant (de la clause). Les environnements de Prolog peuvent être comparés à l'implémentation d'un langage gérant les variables locales (exemple : le C).
- Le backtracking (retour arrière) : qui consiste à reprendre le calcul à la dernière alternative de preuve de l'arbre non encore explorée. On parle alors de point de choix pour désigner cet emplacement, qui a pour fonction de stocker les informations nécessaires à la reprise du calcul. On y retrouve donc la sauvegarde des registres d'arguments et des différents pointeurs, ainsi que l'adresse de la nouvelle clause à essayer.

Dans la WAM originale, qui est étudié ici, ces deux types de blocs de contrôle (environnement et point de choix) sont entrelacés. Le chaînage de ces blocs se fait à l'aide d'un pointeur entre chaque bloc de même type, vers son suivant. Pour assurer la gestion de ces blocs, il faut prendre en compte deux nouveaux registres, **E** et **B** pointant respectivement sur le dernier environnement et le dernier point de choix dans la pile locale. Lors de l'ajout d'un nouveau bloc sur la pile locale, sa nouvelle adresse est obtenue facilement en calculant le maximum entre E et B. Cette approche en pile mixte permet de synchroniser les environnements et les points de choix pour gérer aisément le backtracking. Ainsi, si un environnement est libéré alors qu'il existe un point de choix créé au-dessus de lui (par un de ses fils), l'espace de cet environnement n'est pas réutilisé car il sera nécessaire lors du backtracking.

Voici maintenant en détail le contenu d'un environnement et d'un point de choix tel que figurant dans la wam traditionnelle `[wam1]`.

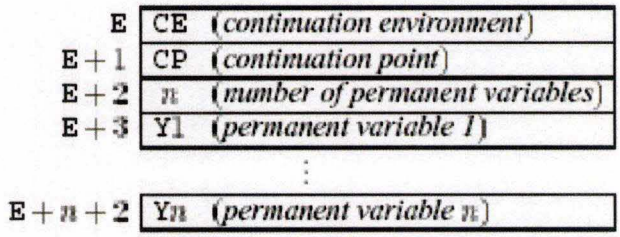

Figure 6 - Description d'un environnement [wam1]

Le champ **CE** constitue le lien de chaînage des environnements, il désigne l'environnement précédent sur le stack. **CP** désigne quant à lui l'adresse de l'instruction suivante à exécuter lorsque la clause définie par l'environnement aura été satisfaite ; il s'agit donc de l'adresse de retour lorsque la clause sera terminée. L'on trouve ensuite le nombre de variables permanentes[1] qui devront être gérées par l'environnement et viennent enfin le contenu de ces variables dans les cellules adjacentes ; elles sont généralement notées Y[1] … Y[n].

[1] Par opposition aux variables dites temporaires qui, au sein d'une clause, sont celles qui ne feront jamais l'objet d'un appel à un prédicat. Celles-ci peuvent donc être traitées par les registres globaux et donc ne pas figurer dans l'environnement.

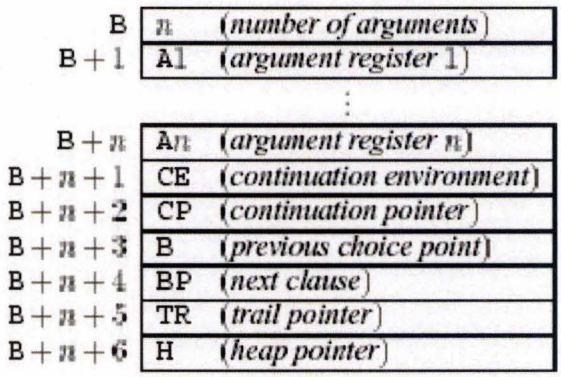

Figure 7 - Description d'un point de choix [wam1]

Un point de choix est d'abord constitué d'un champ représentant le nombre d'arguments qui sont stockés dans les cellules suivantes. Après l'information concernant les arguments, le champ **CE** contient l'adresse du dernier environnement du stack. **CP** a la même signification que dans l'environnement, il y est juste recopié depuis l'environnement courant. Le champ B est le pointeur vers le point de choix précédent sur le stack. Le champ **BP** contient l'adresse de la prochaine clause à essayer si le choix effectué n'est pas le bon. Si la clause qui fait l'objet de ce point de choix n'est pas prouvable, l'exécution se poursuivra avec la clause dont l'adresse est contenue dans BP. Ensuite, **TR** est un pointeur vers le TRAIL (qui est abordé dans le point suivant), qui sauvegarde la référence du sommet de cette pile lors de la création de ce point de choix. Il en va de même pour le champ **H** qui conserve le sommet de la pile HEAP au moment de la création du choice point. Cette référence sera utile en cas d'échec et donc de backtracking : si des variables ont été ajoutées au HEAP après ce point de choix, elles seront perdues lorsque l'on restaurera le sommet de la pile à la valeur de la sauvegarde de H. Enfin le dernier champ, **BO** est un registre permettant d'assurer la gestion des instructions modifiant le déroulement d'un programme Prolog (cut !). La gestion de ces instructions n'est pas traitée dans ce mémoire. Pour une description plus détaillée du cut, l'on peut se référer à [WAM1] (p. 83) et pour les différentes manières d'optimiser leur gestion il faudra consulter [WAM2] à la page 121.

2.4 Gestion des substitutions

L'unification, comme je l'ai présenté plus haut, génère des substitutions entre des variables et des termes. Ces substitutions doivent être gérées de manière à être sauvegardée d'une clause à l'autre (environnements) et à pouvoir être défaites lors d'un backtracking. Cette gestion s'appuie sur une nouvelle pile dans l'architecture de la WAM, allée TRAIL.

Reprenons le formalisme de l'arbre de preuve et des états de recherche, illustré par la Figure 5. Rappelons les états $< r, \sigma, r >$, dont σ est un vecteur associant chacune des variables locales de la clause à sa valeur. En regardant à nouveau la Figure 5, l'on constate que certaines substitutions affectent des variables n'apparaissant pas dans la clause courant, mais dans un environnement antérieur (ces substitutions sont représentées par une flèche →). Ceci pose un nouveau problème : lors du backtracking, si l'environnement modifié est antérieur au dernier point de choix, il faudra défaire ces liaisons pour pouvoir relancer le calcul sur une autre alternative, avec les données d'origines, non perturbées par l'état de recherche qui a modifié cet environnement. Sur la Figure 5, nous pouvons remarquer, au sujet de la liaison de A avec 0 (branche de gauche

du point de choix), que le fait d'écrire cette liaison à cet endroit indique bien que lors du backtracking, nous voulions « oublier » cette liaison. Pour preuve, dans la branche de droite A est lié à s(X3).

La solution pour réinitialiser de telles variables est d'introduire cette pile TRAIL, qui lorsqu'une variable antérieure au dernier point de choix doit être liée, contiendra la référence de cette variable. Ainsi, en conservant dans les points de choix le sommet de cette pile (TR), il suffit lors du backtracking de remettre à l'état « libre » toutes les variables référencées par le trail entre le sommet actuel et le sommet enregistré dans le point de choix.

Enfin, il existe une petite optimisation qui permet de limiter l'empilement dans le trail. Lorsque deux variables doivent être unifiées, une des deux sera obligatoirement de la clause courante ; il faudra donc veiller à ce que cette variable enregistrer la liaison. D'une manière générale, l'on peut reprendre la règle présente dans [WAM4] : « *La liaison entre deux variables s'effectue toujours de la plus récente vers la plus ancienne, i.e. de celle d'adresse la plus grande vers celle d'adresse la plus petite.* »

2.5 Récapitulatif des registres utilisés

Voici les différents registres globaux utilisés par la machine abstraite :

P	pointeur courant sur le code du programme
CP	pointeur de continuation, lors du retour de l'appel à un prédicat
S	en cas de lecture d'un terme à unifier, contient toujours l'adresse dans HEAP du prochain sous terme qui devra être mis en correspondance. Pointeur sur structure à décomposer.
H	pointeur vers le sommet du tableau HEAP
HB	contient la valeur du registre H avant le dernier point de choix.
E	pointeur sur l'environnement courant
B	pointeur sur le point de choix courant
BO	pointeur sur point de choix en cas de coupure
TR	pointeur sur le sommet du TRAIL
A[i]	registres d'arguments globaux

Enfin, pour terminer ce chapitre consacré à la machine abstraite du Prolog, voici un exemple de compilation en code WAM à partir d'un programme Prolog qui est issus de [WAM2].
Le jeu d'instructions de la WAM peut être consulté en annexe 1.

Exemple 12 - Résultat d'une compilation

```
reverse(L, M) :- rev(L,M, []).
rev([], L, L).
rev([A|L], Y, Z) :- test(A), rev(L,Y, [A|Z]).
```

Voici le résultat en instructions WAM d'une compilation possible :

```
reverse/2                          % procedure pour reverse/2
                                   % les arguments d'entrée sont dans A0 A1
        put_nil A2                 % charge le registre A2 avec []
        call rev/3                 % appelle rev/3 avec les arguments dans A1 A2 A3
        proceed                    % retour réussi pour reverse/2
```

```
rev/3

        try_me_else clause2
    clause1 :
        get_nil A0
        get_x_var  X3 A1
        get_X_value  X3 A2
        proceed
    clause2 :
        trust_me_else_fail
        allocate  4

        get_list  A0
        unify_y_variable  Y1
        unify_y_variable  Y3
        get_y_variable    Y2 A1
        get_y_variable    Y0 A2
        put_y_value       Y1 A0
        call test/1
        put_y_value       Y3 A0
        put_y_value       Y2 A1
        put_list            A2
        unify_y_value       Y1
        unify_y_value       Y0
        call rev/3
        deallocate
        proceed
```

```
% procédure pour rev/3
% les arguments sont dans A0 A1 A2
% encas de d'échec dans la clause 1, aller à clause 2
% label marquant la clause 1 de rev/3
% unifie A0 avec []
% unifie A1 avec la variable temporaire X3
% unifie A2 avec le contenu de X3 (donc A1 = A2)
% retour réussi
% label marquant la clause 2 de rev/3
% pas d'autre alternative après clause2
% alloue 4 variables Y0 Y1 Y2 Y3
% pour                  Z  A  Y  L
% unifie A0 avec une liste [
%                            Y1 |
%                               Y3]
% unifie Y2 avec A1
% unifie Y0 avec A2
% charge A0 avec le contenu de Y1
% appelle test/1 argument dans A0
% charge A0 avec le contenu de Y3
% charge A1 avec le contenu de Y2
% charge A2 avec la liste [
%                            Y1 |
%                               Y0]
% appelle rev/3 arguments dans A0 A1 A2
% désallocation de Y0 Y1 Y2 Y3
% retour réussi
```

3 Bin – Prolog

Je vais maintenant exposer une variante du langage Prolog, qui permet une simplification de l'implémentation du Prolog, qui permet de modifier plus facilement le code du langage. Il s'agit de Bin Prolog, un sous-ensemble du Prolog auquel correspond une machine abstraite BinWAM. Celle-ci est plus exactement un moteur émulant le Prolog, développé en C ; tel que le définit son créateur Paul Tarau dans [bin1].
Ce moteur se base sur une transformation du Prolog qui vise à transformer un programme Prolog vers un programme à logique binaire.

3.1 Mise sous forme binaire

Cette transformation consiste en un pré-processeur qui applique la technique de binarisation au programme prolog qui lui est fournit. Cette technique permet d'obtenir un programme à logique binaire, où il ne peut y avoir qu'un seul but par règle.

Voici un exemple de binarisation d'une procédure Prolog.

Exemple 13 - Binarisation d'une procédure

Source clause p(X) :- q(X), r(X,Y), s(Y).
Binary clause :- p(X, Cont) :- q(X, r(X,Y, s(Y, Cont)).

Source clause : append ([], Ys, Ys) .
Binary clause : append ([], Ys, Ys, Cont) :- true(Cont).

Source clause : and(X, Y) :- X, Y.
Binary clause : and(X,Y, Cont) :- call(X, call(Y, Cont)).

Où les contrôles AND sont transformés en termes (ce qu'on appelle la Continuation dans la littérature consacrée au sujet).

Cette mise sous forme binaire, permet de supprimer la gestion des environnements au sein de la machine abstraite. Cette suppression est une conséquence de la binarisation dans le fait où aucun retour n'est nécessaire, en effet une fois la clause q appelée (dans l'exemple précédent), le règle p est terminée, il ne faut donc pas ramener les infos de q pour appeler une clause suivante. Il n'y a donc qu'un seul environnement global dans la BinWam, et les variables sont toutes temporaires, dans les registres globaux.

On peut donc remarquer que la binarisation supprime l'utilisation des environnements dans le stack de la wam, et que la continuation est ajoutée sur le tableau heap, en étant inscrite de manière récursive à l'aide du dernier argument qui est ajouté à chaque prédicat.

Pour une étude très précise de la binarisation des programmes logiques, il est conseillé de consulter [bin4] concernant la binarisation des méta-programmes.

3.2 Bin Wam

Tout comme le pour le Prolog, le bin Prolog est un langage qui doit être interpréter à l'aide d'une machine virtuelle. Cet interpréteur porte le nom de BinWam (binary wam).

Cette machine virtuelle ne dispose plus, dans son stack, des environnements permettant de sauvegarder son état courant avant les appels de prédicats. Comme nous l'avons vu plus haut, ceux-ci deviennent inutiles, suite à la binarisation des procédures.

La bin wam nécessite toujours la présence des points de choix, ou « Or stack ». Le contenu de ces points de choix est également simplifié. En effet, il ne faut plus garder les pointeurs vers l'environnement courant. La figure en présente le contenu.

P ⇒	next clause address
H ⇒	backtrack top of the heap
TR ⇒	backtrack top of the trail
A_1 ⇒	saved argument register 1
...	...
A_N ⇒	saved argument register N

Figure 8 - Point de choix dans la BinWam

Suite à la suppression des environnements dans l'architecture de la bin wam, le jeu d'instructions est simplifié des instructions manipulant les variables (Y variables), ainsi que des instructions d'allocation et de désallocation des environnements.

3.3 Evaluation

Nous allons maintenant lever les avantages et inconvénients reconnus du BinProlog
Tout d'abord, l'absence d'environnements dans la machine abstraite du langage (BinWam), permet une simplification de l'implémentation du langage. La gestion des environnements est donc supprimée, ce qui réduit le nombre d'instructions de la machine ainsi que celui des données sauvegardées dans les points de choix. En effet, il n'est plus nécessaire de garder la valeur du registre E dans le champ CE.
Cependant, en retour, cette simplification à un prix, représenté par une plus large consommation de la pile mémoire, HEAP. Mais de nombreuses

On peut constater également certaines différences par rapport au Prolog concernant les instructions et la rapidité de ce langage.
Tout d'abord, le jeu d'instructions de la BinWAM est réduit, ainsi les instructions concernant les environnements son retirées. Par contre, son espace Heap est quant-à lui supérieur à celui utilisé par la WAM classique.
Le noyau de la BinWAM est réduit à 23 instructions élémentaires de la WAM.

En plus de simplifier l'implémentation de sa WAM le Bin Prolog permet également des transformations de code plus aisées, ce qui pourra être utilisé si l'on désire se pencher plutôt sur la solution qui consiste à augmenter le langage, ici le Bin Prolog.

L'on parle également du multi BinProlog, comme étant un BinProlog gérant le multi-threading. Cette fonctionnalité, qui permet de gérer l'exécution parallèle de plusieurs processus, permet donc d'exécuter plusieurs programmes prolog sur la même BWAM.

Chapitre 2 : Web – CGI

Maintenant que les bases du langage Prolog et de ses implémentations sont posées, je vais parler du monde internet. Internet -- très vaste réseau de machines interconnectées – représente un très large moyen de diffusion d'information. Cette information et surtout sa présentation est de plus en plus dynamique, c'est à dire donc le contenu change quotidiennement, voire plusieurs fois par jour. Cette dynamique de l'internet est assurée par des programmes externes qui génèrent une page html qui est alors envoyée au navigateur de l'utilisateur final. Il faut donc pouvoir interfacer le coté web, (html) avec ces applications externes. Voici une manière possible de dialoguer entre un serveur web, quel que soit le langage utilisé pour les applications de ce serveur.
Il s'agit de la technologie CGI (Common Gateway Interface) qui est une interface entre des applications externes et des serveurs d'informations tels que des serveurs WEB.

Un document HTML peut être une page statique ou une page dynamique si elle est le résultat du travail d'un programme utilisant CGI. Par abus de langage, un script utilisant l'interface CGI est appelé script CGI ou directement CGI.

1. Fonctionnement.

Le navigateur WEB communique avec le serveur hôte, appelé Daemon, via le protocole HTTP (Hyper Text Transfert Protocol). Quand un navigateur Web appelle un CGI qui accède à une BD, le serveur exécute le CGI. Le CGI exécute la requête et communique avec le moteur de base de données via les API du SGBD. Le SGBD retourne au CGI les données demandées, qui va les formater en HTML et les renvoyer au serveur. Le serveur va les renvoyer au navigateur qui les a demandées. Le navigateur va alors formater les pages reçues de façon à en permettre l'affichage. Les CGI ne servent pas qu'à accéder à des bases de données, ils permettent d'automatiser certaines opérations comme l'ajout de commentaires dans un livre d'or par exemple.

La figure suivante illustre l'appel à un programme via l'interface CGI :

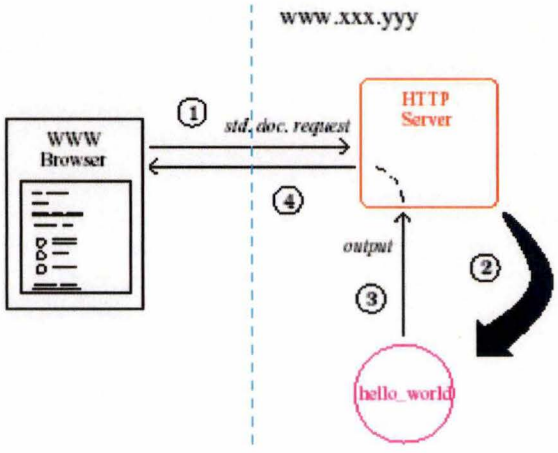

Figure 9 - exemple d'appel CGI

Le navigateur client lance une requête CGI au serveur http. Celui-ci lance le programme qui y est associé, avec les arguments éventuels (via l'entrée standard) et récupère le résultat de l'exécution (via la sortie standard). Enfin, il renvoi ce résultat vers le navigateur, pour que ce dernier puisse les afficher à l'utilisateur.

Les CGI peuvent être écrits en plusieurs langages : Perl, PHP, ASP,... sont les plus courants. Le CGI n'est donc pas un langage, mais un moyen de d'interagir entre le serveur web et le client. Lorsqu'un formulaire html est envoyé, c'est CGI qui gère l'envoi des données vers le fichier sur le serveur qui va les traiter (fichier php, perl,...). Cet envoi s'effectue à l'aide de deux méthodes soit POST soit GET.

- **La méthode GET**

  Quand on utilise cette méthode, le programme reçoit les données dans la variable d'environnement QUERY_STRING.
  Le programme doit traiter la chaîne d'entrée pour être en mesure d'interpréter les données et d'effectuer les actions appropriées. La méthode GET ne peut être utilisée que si les données d'entrées ne sont pas trop importantes; car la longueur de la variable d'environnement QUERY_STRING est limitée à 1024 caractères.

- **La méthode POST**

  Quand on utilise cette méthode, les données à traiter sont transmises via l'entrée standard (STDIN). Le serveur n'indique pas la fin de la chaîne avec un caractère, c'est pourquoi il faut utiliser la variable d'environnement CONTENT_LENGTH pour pouvoir lire les données correctement.

  Ces deux méthodes font parties des trois moyens de communiquer définies par le CGI, ils sont décrits dans les trois points suivants.

2. Les variables d'environnement

Ce sont des variables propres au système CGI. Elles sont initialisées par le serveur WEB quand il exécute un programme CGI. Je ne vais en citer que quelques-unes, les plus courantes.

- CONTENT_LENGTH

  Cette variable donne la longueur, en bytes, des données envoyées au CGI quand on utilise la méthode POST. Elle est vide si on utilise la méthode GET.

- CONTENT_TYPE

  CONTENT_TYPE fournit le type MIME des données envoyées au programme CGI appelé par la méthode POST. Si on utilise la méthode GET, CONTENT_TYPE est vide

- QUERY_STRING

  QUERY_STRING contient l'information qui suit le caractère ? Dans une URL qui est envoyée au programme CGI. Quand on utilise la méthode GET, QUERY_STRING contient les données d'entrée du programme. Avec la méthode POST, QUERY_STRING est vide à moins qu'il n'y ait quelque chose derrière l'URL du script (un ? suivi de texte)

- REMOTEADDR

  REMOTEADDR contient l'adresse IP de l'ordinateur qui a effectué la requête. Cette variable permet de repérer, d'identifier des ordinateurs et d'effectuer quelque chose en conséquence (empêcher l'accès, donner des droits supplémentaires par exemple).

- REMOTE_HOST

  REMOTE_HOST permet de connaître le nom de domaine de l'ordinateur qui a fait la requête. Cette variable est fort utilisée pour afficher des publicités en rapport avec le pays d'origine par exemple.

- REQUEST_METHOD

  REQUEST_METHOD permet de connaître quelle méthode a été utilisée : GET ou POST? Cela sert pour connaître la manière dont on va traiter les données.

3. L'entrée standard

L'entrée standard (STDIN) est utilisée par le serveur WEB pour passer des informations au programme CGI quand la méthode POST est utilisée. Le serveur envoie aussi les variables d'environnement CONTENT_TYPE et CONTENT_LENGTH de telle manière que le programme CGI sache quel type de données il reçoit et sur quelle longueur. CONTENT_LENGTH contient la longueur des données quand elles sont sous forme URL-encoded; c'est-à-dire que les espaces sont remplacés par des signes +, les tildes (~) sont remplacés par %7E. Le principe de cet encodage est assez simple : le symbole % est suivi du code ascii du caractère au format hexadécimal. La figure suivante présente le codage des principaux caractères et symboles.

Caractères communs		Symboles	
espace	%20	"	%22
&	%26	#	%23
/	%2F	$	%24
?	%3F	%	%25
!	%21	£	%A3
Caractères spéciaux		'	%27
backspace	%08	(	%28
tab	%09	)	%29
linefeed	%0A	*	%2A
creturn	%0D	+	%2B
		,	%2C
Symboles mathématiques		-	%2D
±	%B1	.	%2E
		:	%3A
·	%B7	;	%3B
×	%D7	<	%3C
÷	%F7	=	%3D
¹	%B9	>	%3E
²	%B2	@	%40
³	%B3	[	%5B
¼	%BC	\	%5C
½	%BD	]	%5D
¾	%BE	{	%7B
¬	%AC	|	%7C
<	%3C	}	%7D
>	%3E	~	%7E
		^	%5E

Figure 10 - Codage URL de principaux caractères et symboles

4. La sortie standard

Le programme CGI envoie les résultats vers la sortie standard (STDOUT soit l'écran en général). Ils peuvent être envoyés directement vers le navigateur du client ou être interprétés par le serveur qui va effectuer une nouvelle action. Les programmes CGI peuvent court-circuiter le serveur WEB et converser directement avec le navigateur.
Dans les résultats renvoyés, le serveur cherche un des 3 en-têtes que le programme peut retourner :

- Content-type : indique le type MIME des données. Généralement comme les programmes CGI renvoient de l'HTML, la ligne utilisée est Content-type: text/html\n\n.

  Il faut mettre 2 nouvelles lignes (\n) pour placer une ligne blanche après l'en-tête HTTP du serveur.

- Location : indique au serveur que l'on fait référence à un autre document.

- Status : c'est le code d'état renvoyé par le serveur au client. Format : nnn XXXXXXX où nnn est un nombre à 3 chiffres et XXXXXX le texte qui y correspond. Exemple : 404 Not found.

Chapitre 3 : Linda & Tableaux

Une autre solution à envisager pour communiquer entre le client web et le serveur Prolog est d'utiliser une couche de coordination. Cette couche peut être ajoutée à n'importe quel langage, comme par exemple Fortran, C et Prolog. Elle fournit un espace global de "tuples" qui sont des cellules étiquetées sur un tableau partagé. On peut accéder aux tuples depuis chaque processus par un petit nombre d'opérations primitives.

Le modèle Linda va d'abord être présenté. Nous terminerons par un exemple de programme utilisant Linda afin de montrer un aperçu du langage lui-même.

1. Définition de Linda

Cette définition du modèle Linda est en grande partie inspiré du cours `[linda1]`.

Le modèle Linda est basé sur une mémoire partagée accessible par plusieurs processus. Cette mémoire partagée est constituée de

- Une collection de tuples notée Ts
- Un ensemble d'opération qui manipulent Ts : comme l'ajout, la suppression et la lecture de tuples.
- Un mécanisme d'unification qui permet aux opérations d'accéder aux tuples de Ts.

a) Définition d'un tuple

Un tuple est une suite finie et ordonnée de champs typés. Un champ peut contenir soit une valeur typée soit un processus.
Si un tuple ne contient que des champs de valeurs typées, il est appelé un tuple de données, sinon c'est un tuple de processus.
Contrairement au tuple de données, le tuple de processus est actif car il échange des données en créant, consultant et supprimant des tuples de données.
Une fois que le tuple de processus a terminé son exécution, il devient un tuple de données.
Enfin, les tuples de processus s'exécutent en parallèle.

Il existe une autre notion : les anti-tuples, qui sont des tuples de données mais dont les champs peuvent être une place libre et non une valeur. Les champs ayant une valeur sont appelés les champs « paramètre valeur » (actual) et les champs vides se nomment « paramètre formel » (formal).
Les anti-tuples sont utilisés par le mécanisme d'unification qui essaie toujours d'unifier un anti-tuple à un tuple.

b) Opérations manipulant les tuples

Parmis les opérations manipulant les tuples, intéressons-nous d'abord à la **lecture**. Il existe deux primitives rd(a) et rdp(a).
Commençons par rd(a).

Si

Il existe un tuple de données t dans l'espace Ts qui s'unifie avec l'anti-tuple a

Alors

Les paramètres valeurs de t sont assignés aux paramètre formels de a et le processus appelant reprend son exécution.

Sinon

Le processus appelant est mis en attente.

Si plusieurs tuples peuvent s'unifier avec a, l'unification est choisie au hasard ; ce choix est supposé équitable. Il en va de même si plusieurs anti-tuples en attentes peuvent s'unifier avec un tuple t.

Pour rdp(a) : l'appel est équivalent mais non bloquant. Si l'unification ne peut avoir lieu, rdp retourne la valeur 0 et le processus appelé reprend son exécution.

Ensuite, la suppression. Il existe également deux primitives in(a) et inp(a).
Commençons par in(a).
Le fonctionnement de in(a) est identique à celui de rd(a) mais le tuple unifié t est retiré de Ts.
Il en va de même pour inp(a) : si l'unification a pu avoir lieu, le tuple t est retiré de Ts.

Enfin, l'ajout. Il existe deux primitives out(t) et eval(t)
Avec out(t) le tuple t est évalué puis ajouté à Ts. Le processus appelant reprend son exécution.

Pour eval(t), un processus p qui permet d'évaluer le tuple t est créé. Une fois p créé, le processus appelant reprend la main : p s'exécute en parallèle avec les autres processus du système.

c) Mécanisme d'unification
Malheureusement le modèle Linda défini par leurs créateurs, Carriero et Gelernter, ne précise pas le mécanisme d'unification. Ce procédé doit être défini lors de l'implémentation de Linda dans un langage donné.

Il est à noter qu'il n'existe pas d'instruction de modification des tuples dans ce modèle, contrairement aux bases de données. Cette instruction n'est pas nécessaire de par la nature de ce formalisme. Son but n'est pas de garder des données de manière persistantes et à y fournir un accès, contrairement aux bases de données qui ont un rôle d'archivage et où la modification a un sens en soi. Il s'agit ici de mettre à disposition des processus, d'un espace partagé servant de coordination entre eux. Si un processus doit modifier la valeur d'un tuple, il procédera en deux phases : la suppression de ce tuple, puis l'ajout d'un tuple avec de nouvelles valeurs.

2. Exemple de programme utilisant Linda

Est-il encore besoin de le présenter le bon vieux dîner des philosophes ? Autour d'une table ronde comprenant n assiettes, une baguette entre chaque paire d'assiette et un bol de riz au milieu. Les n philosophes sont dans un cycle infini « penser – manger ». Pour manger il faut avoir deux baguettes. Une fois le bol vide, le philosophe repose ses baguettes pour permettre à ses voisins de manger.

Voici une solution en pseudo-code Pascal-Linda, extraite de [linda1]

Exemple 14 - problème des philosophes avec Linda

```
% création de n baguettes, n philosophes et n-1 tickets
for i := 1 to n do
        out("baguette", i);
        eval(philosophe(i));
        if i < n then out ("ticket") end if;
end for;

% définition d'un philosophe
procedure philosophe (i:integer);
begin
        loop forever
                penser;
                in("ticket");
                in("baguette", i);
                in("baguette", (i+1)div n);
                manger;
                out("baguette",i);
                out("baguette", (i+1) div n);
                out("ticket");
        end loop;
end philosophe;
```

La présence de tickets assure l'absence d'inter blocage. Pour manger un philosophe doit avoir un ticket et il n'y en a que n-1. Tous ne peuvent donc vouloir manger en même temps.

L'équité et l'absence de famine sont assurées par le procédé d'unification qui est garantit par leur créateur comme étant aléatoire et équitable.

Les tickets, les philosophes et les baguettes sont géré par Linda et sont donc représentés dans un Ts partagé par les différents processus représentant chacun un philosophe.

Linda et les tableaux partagés peuvent être une bonne approche pour communiquer entre les serveurs web Prolog et le client web navigateur.

Partie II : Applications

Cette deuxième partie, aura pour but dans un premier temps de présenter les agents intelligents qui sont les composants qui doivent être interfacés par la librairie qui fait l'objet de ce mémoire. Ensuite pour terminer cette seconde partie, je parlerai de deux applications qui sont interfacées avec l'utilisateur à travers Internet.

Chapitre 4 : Traitement langage naturel

L'intégration du langage naturel permet bien des perspectives : il peut ouvrir la voie vers le contrôle vocal du monde virtuel, il est également une base pour mettre en œuvre la reconnaissance/génération vocale pour l'interaction homme machine. Il est déjà utilisé par certaines applications comme une interface homme-machine à part entière. Je vais prendre ici le cas de l'application LogiMoo (qui sera analysée au point « 6.2. LogiMoo », à fin de relever les caractéristiques majeures du traitement du langage naturel.

1. Langage Naturel pour l'Interface Homme Machine de LogiMoo

LogiMOO est un des rares mondes virtuels qui peut être contrôlé à l'aide du langage naturel écrit. Un monde virtuel est un excellent environnement pour expérimenter le langage naturel car le domaine de discussion est limité. Ici aux opérations à effectuer (créer, déplacer des objets,...).

Voici maintenant la présentation de l'étendue du langage que peut comprendre LogiMOO.
Tout d'abord, les phrases utilisées pour contrôler ce monde virtuel sont généralement à l'impératif, ce qui entraîne que l'avatar (le sujet) est implicite et ne doit donc pas être mentionné dans la phrase.
Grâce à cette utilisation de l'impératif les phrases sont réduites à des formes verbales voici leurs descriptions.

- Un verbe intransitif.
- Un verbe transitif puis un nom.
- Un verbe transitif puis une forme nominale.
- Un verbe transitif puis une phrase prépositionnelle.
- Un verbe bi transitif puis deux formes nominales.
- Un verbe bi transitif puis une forme nominale et une phrase prépositionnelle.

On peut définir ensuite la phrase prépositionnelle :

- Une préposition puis une forme nominale

La forme nominale :

- Un nom propre
- Un pronom
- Un déterminant puis un nom

À cela viennent s'ajouter les instructions de communication, que l'on peut définir comme suit (en anglais):

- *Whisper* puis une phrase prépositionnelle puis un message
- *Say* puis un message

- *Yell* puis un message

Ces règles et définitions sont utilisées par le parseur afin de traduite ces phrases en instructions LogiMOO. Voici un exemple de traductions avec les effets obtenus (les exemples sont en anglais car le langage naturel géré par LogiMOO est en anglais) :

Langage naturel	**Traduction LogiMOO**	**Action effectuée**
Look	Look	Fournit une description de la pièce et des avatars présents
craft a car	craft(car)	Crée un objet virtuel, car, possédé par l'avatar
craft a car and give it to john	craft(car) give(john, car)	Crée un objet, car, et le donne à john
take the car that john crafted	take(car)	Met le car possédé par john dans les possessions de l'avatar

Concernant la dernière traduction, il est nécessaire de savoir quel nom porte l'objet désigné par la phrase. Pour le déterminer, le parseur consulte la base de connaissances du monde virtuel. Il n'existe pas d'ambiguïté quant à la voiture qu'il faut prendre parmi celles présentes. S'il y a plusieurs objets « car », ils sont différents quant à leur représentation en interne dans LogiMOO, dépendant de leur propriétaire et de la place où ils se trouvent.

Les mondes virtuels étant très changeants, il faut gérer correctement et légèrement ces changements. La base de connaissances du monde se découpe en deux types : les connaissances statiques et les connaissances dynamiques. La première concerne la connaissance du monde avant une séquence d'instructions, de l'utilisateur, modifiant ce monde. La seconde sont les connaissances qui résultent de ces instructions car elles peuvent n'être que des tentatives et être revues, corrigées, modifiées. Les connaissances statiques sont sauvegardées avant de parser les instructions en langage naturel en sauvegardant l'état courant à l'aide de prédicats tels que ceux-ci :
isavatar(X), is_crafteed(X) ...

Les connaissances dynamiques sont crées lors de l'exécution de ces instructions à l'aide des mêmes prédicats mais, où ils sont enregistrés dans un autre tableau au lieu d'étendre les connaissances statiques. Une fois les instructions effectuées (avec modifications éventuelles), ce tableau est rendu disponible en vue d'être assemblé aux données statiques avant la prochaine série d'instructions en langage naturel.

Il existe encore un autre type de connaissances utilisées par le parseur, il s'agit du savoir hypothétique. Il est utilisé pour déterminer les pronoms et les noms auxquels ils se rapportent. Prenons comme exemple l'analyse de cette commande en langage naturel : « craft a flower, give it to john », le parseur émet comme hypothèse que « flower » pourra être utilisé plus tard pour être désigné par un pronom. Quand il rencontre « it », il associe le bon nom avec ce pronom pour traiter cette instruction. Ces connaissances comprennent également d'autre type de données telles que le nombre, le genre pour permettre d'identifier ce type de relations.

Chapitre 5 : Agents intelligents – Prolog

Avec l'expansion fulgurante du contenu présent sur internet et de sa complexité, la technologie des agents intelligents arrive au bon moment pour apporter sa contribution dans la gestion de cet ensemble désordonné de pages. Ce chapitre va traiter de la technologie agent et brièvement de son application dans le réseau internet. Il a pour but de montrer l'interaction entre les agents à travers d'un réseau et de présenter brièvement les protocoles existants quant à la communication inter-agents.

1. Définitions

Avant tout comment pouvons-nous définir un agent intelligent ?
Commençons par la notion d'agent, en laissant momentanément la notion d'intelligence de coté.
Le dictionnaire (Larousse) donne une définition du mot agent : du latin 'agens', celui qui agit.
« un agent est une personne chargée des affaires et des intérêts d'un individu, d'un groupe ou d'un pays, pour le compte desquels elle agit. »
Et agir prend le sens de « faire quelque chose, de produire un effet ».

En mettant ensemble les diverses définitions rencontrées sur le sujet, je définirai un agent logiciel comme ceci :

« Entité informatique qui réalise de manière autonome une tâche à la demande d'un utilisateur ou d'un autre agent ; où la tâche est une activité, suite de fonctionnalités offerte par son environnement».

Passons maintenant à la notion d'intelligence que nous avons mis de coté. A mes yeux il s'agit de la *faculté de comprendre et surtout de s'adapter à des situations nouvelles, à une modification de son environnement.* Cette définition qui vaut également pour l'être humain me semble appropriée pour le sujet qui nous préoccupe.

Les grandes caractéristiques de l'intelligence se trouvant sur le site de « Décisionnel » consacré à l'intelligence artificielle me semblent pertinentes pour définir l'intelligence de ces agents. Nous pouvons reprendre ce schéma plaçant un agent intelligent comme une entité pouvant apprendre, communiquer et être fortement autonome.

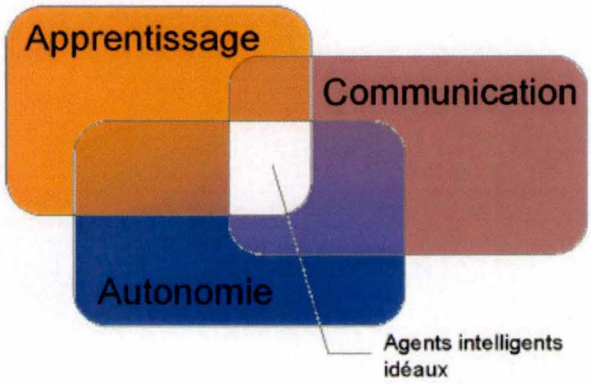

Figure 11 - caractéristiques de l'intelligence

Voici également l'explication de chacune de ces dimensions, caractéristiques de l'intelligence : *Composants logiciels et / ou matériels capables à des degrés différents* `[agent1]` *:*

D'être ***hautement*** *autonome*	*L'agent peut fonctionner de manière autonome c'est à dire sans intervention directe humaine ou autre, et de plus il a une forme de contrôle sur ses actions et sur son état interne.* *Par ex : un agent réseau peut décider, de lui-même, lorsqu'il est inactif, de procéder à des statistiques sur les routeurs en vue d'améliorer son activité future*
D'apprendre	*Un agent aura la capacité d'apprendre s'il sait acquérir de la connaissance et/ou des réactions.* *Les agents doivent être capables de réagir avec un environnement, de s'adapter aux circonstances, de prendre une décision ou d'enrichir eux-mêmes leur propre comportement, sur la base d'observations qu'ils effectuent.* *Par ex : si un agent doit se déclencher à une certaine heure (réactivité)mais que l'utilisateur le coupe car il ralentit cet utilisateur, l'agent doit pouvoir apprendre à différer son exécution pour éviter de le gêner à nouveau*
De communiquer	*Appelée aussi la capacité sociale, la communication est l'interaction de l'agent avec d'autres agents (potentiellement des être humains) grâce à des interfaces de communication inter-agents. Il s'agit d'une base fondamentale pour la coopération.* *Par ex : cet agent de sauvegarde rencontre un autre agent de sauvegarde sur un réseau. Ces deux agents peuvent se partager le travail afin de terminer plus rapidement cette tâche. Ce partage nécessitera une entente et donc de la communication entre eux*

2. Systèmes multi-agents

Afin d'améliorer la productivité d'un agent, par rapport à son objectif, il doit être envisagé d'amener ce dernier à collaborer avec d'autres agents. Cette association entre agents, permet également de faciliter leur conception. En effet, il est plus aisé de concevoir un agent qui ne réalise qu'un objectif de moyenne importance plutôt qu'un énorme agent qui résoudrait tout un problème à lui tout seul. On a recours alors à la programmation multi-agent ; où un objectif complexe est subdivisé en plus petits, et chaque agent réalise un seul de ces objectifs.
Les agents sont donc amenés se coordonnés, se demander des services. Ces réseaux d'agents permettent d'offrir une meilleure modularité, flexibilité et extensibilité à la résolution d'une tâche importante.

La communication entre agents intelligents, qui devient indispensable dans le cadre de ces systèmes multi-agents, est assurée par des langages de communication. Parmi les plus utilisés, on retrouve KQML et ACL. Ces deux langages sont basés sur des échanges de messages entre les agents. Ceux-ci interprètent les messages pour effectuer les actions demandées. Il y a donc une la nécessité de mettre au point des ontologies afin de partager la même sémantique des messages.
Ces langages, constituant un lien entre deux agents, me semblent intéressants à considérer dans ce mémoire car ils constituent un moyen de communication entre agents.

2.1 KQML

De l'anglais « Kwowledge query and manipulation language », KQML est issu d'un groupe de travail du knowledge sharing effort, une commission dont les objectifs étaient le développement d'infrastructures pour le partage de connaissances entre différents systèmes.

Monsieur Bourdon dans son cours sur les systèmes multi-agents `[agent3]` nous définis en quelques mots, l'idée générale de ce langage. *« Le principe de KQML est de séparer la sémantique liée au protocole de communication (indépendante du domaine d'application), de celle liée au contenu des messages dépendante du domaine d'application). Le protocole de communication doit être universel (pour tous les types d'agents), concis et constitué d'un nombre restreint de primitives de communication. »*

Le langage KQML est divisé en une syntaxe à trois niveaux : la communication, et le contenu. Les deux figures qui suivent illustrent d'abord la syntaxe des messages de ce langage. La seconde figure est un exemple.

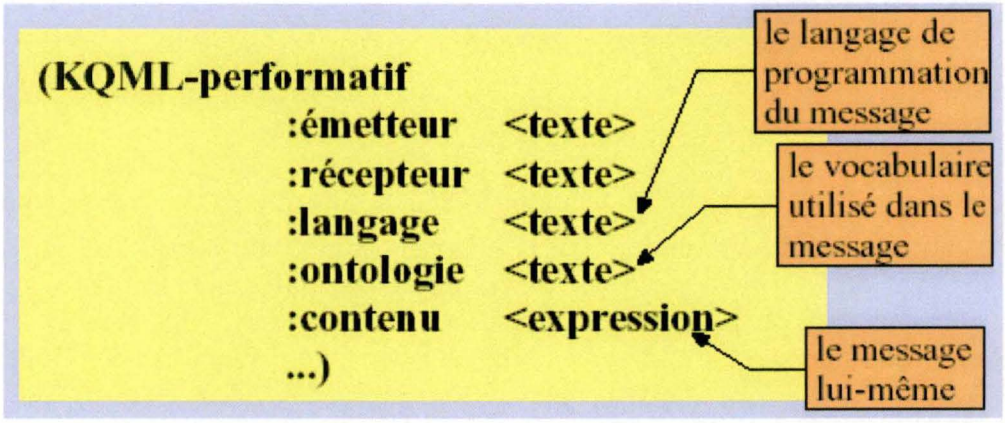

Figure 12 - syntaxe des messages KQML [agent3]

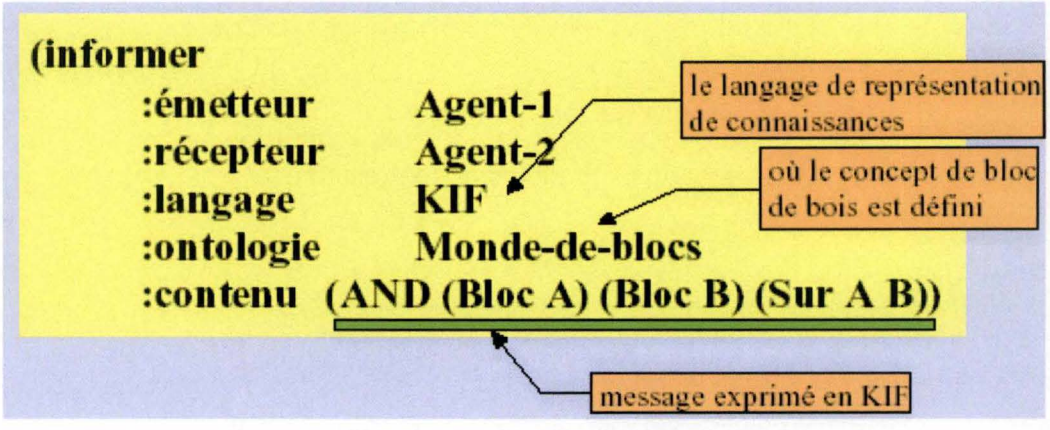

Figure 13 - exemple de message KQML [agent3]

A partir de ces messages, les agents peuvent communiquer entre eux et partager la même signification de leurs messages grâce à un même langage et une même ontologie.
Il me reste à parler d'une technique qu'utilise KQML qui me semble très pratique. Il s'agit de l'imbrication de messages. Cette imbrication permet de relayer un message à travers un agent intermédiaire en imbriquant le message en temps que contenu. Ainsi, pour prendre un exemple concret, imaginons trois agents 1, 2 et 3. L'agent 1 est connecté à 3 et 3 est connecté à 2, mais il est impossible que 1 puisse entrer directement en communication avec 2. L'agent 1 va alors envoyer un message à 3, lui demandant de transférer le contenu de ce message à 2. La figure suivante illustre cet exemple.

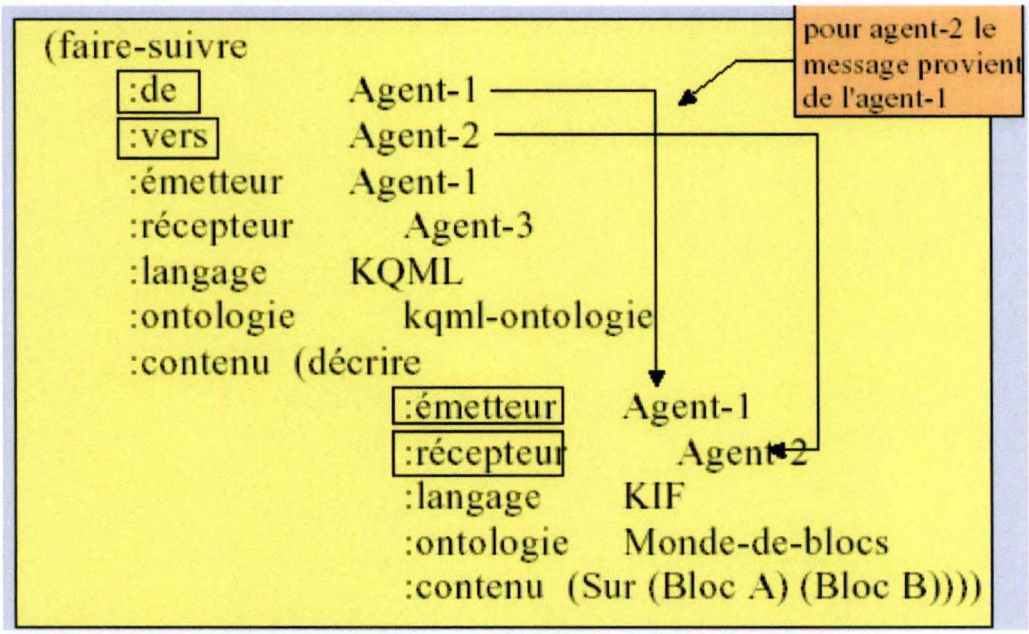

Figure 14 - messages imbriqués en KQML [agent3]

Il existe différents performatifs de messages utilisés par ce langage.

Cependant, ce langage est largement critiqué par les développeurs de systèmes multi-agents. En voici les principales :

- Ambiguïté et imprécision des performatifs : car le sens est exprimé en langage naturel qui est donc laissé au jugement du développeur.
- Certains performatifs sont inutiles et incohérents : soit ce ne sont pas des actes de langage, soit ils satisfont des buts appartenant à d'autres agents, ce qui constitue une perte de productivité (ex : le relais de messages).
- Manque de formalisation et de définition : impossibilité de composer de nouveaux performatifs à partir d'existants.
- Nécessite le partage d'une sémantique commune au niveau des performatifs, même si les agents sont hétérogènes : ceci à fin de créer un système ouvert où les performatifs serviraient d'interface commune.

Pour remédier à ces lacunes, une association, la FIPA, développe un autre langage ACL.

2.2 ACL

La FIPA (Foundation for Intelligent Physical Agent) vit le jour en 1996 et comptait 50 membres à ces débuts. Cette association américaine, qui a déjà de nombreuses spécifications à son palmarès, a pour principal objectif de spécifier tout ce qui a trait :

- A la gestion des agents,
- La communication entre agents,
- L'intégration d'agents

En 1997, cette association spécifie ACL (Agent Communication Language), basé sur une syntaxe similaire à celle de KQML, mais qui définit :

- Un ensemble d'actes de communication de base.
- Une sémantique détaillée de ces primitives.
- Un ensemble de messages prédéfinis que tout agent doit être capable de traiter :
  - "not-understood" : si l'agent reçoit un message qu'il ne peut pas comprendre. Tout agent doit être capable de traiter un tel message.

Ce langage est composé des primitives suivantes :

Information
(contient des propositions)
query_if
query_ref
subscribe
inform
inform_if
inform_ref
confirm
disconfirm
not_understood

Task distribution
(contient des actions)
request
request_whenever
cancel
agree
refuse
failure

Negociation
(contient des actions et des propositions)
cfp
propose
accept_proposal
reject_proposal

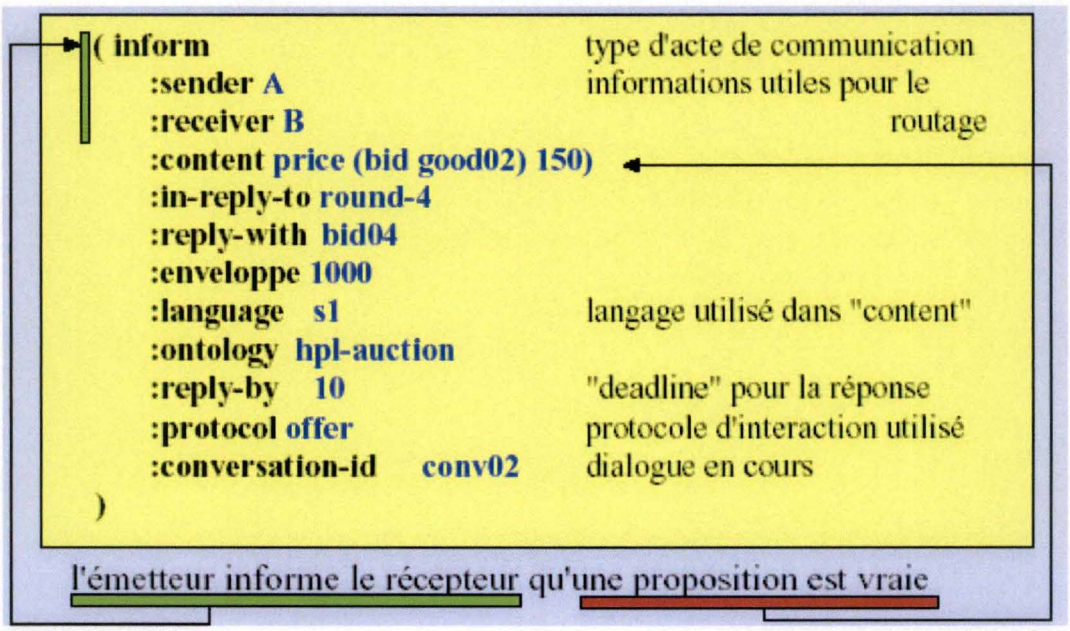

Figure 15 - Exemple de messages ACL [agent3]

Grâce à l'exemple situé ci-dessus, l'on peut constater que le principe de KQML est gardé : l'on retrouve la référence au langage utilisé ainsi qu'à l'ontologie partagée. L'avantage de ce langage

est qu'il est beaucoup plus définit donc, beaucoup plus ouvert afin de supporter des systèmes multi agents hétérogènes.

3. Messages et tableaux

Pour être complet, il reste un dernier point à aborder concernant la différence entre ces langages de communication entre agents au sein d'un système multi-agents(ACL, KQML) et les modèles de coordination par mémoire partagée (Linda). Les premiers dialoguent à l'aide de messages et les seconds à l'aide d'un tableau partagé.

Or, ces deux procédés ne sont pas identiques.

D'abord les messages sont adressés. Nous entendons par là qu'ils viennent d'un agent précis, pour atteindre un autre agent ciblé. Tandis que l'envoi d'une donnée sur un tableau partagé n'est destinée à aucun agent en particulier, la cible est simplement la zone partagée.

Ensuite, une autre différence réside dans la consultation des données échangées. Un agent communiquant par messages doit se tenir en permanence à l'écoute des messages sur le canal de communication. À l'inverse, les processus utilisant une zone partagée ne la consultent qu'à un moment précis pour voir si une certaine donnée est présente ou non, et le cas échéant, attendent ces données.

Enfin, lors de la consultation du message, celui-ci n'est pas visible par les autres agents du système. Par opposition, un processus travaillant par mémoire partagée peut décider soit de laisser la donnée sur le tableau, soit de la supprimer. La technique des messages possède donc un caractère privé, et celles des tableaux partagés un caractère public.

4. Agents mobiles

La dernière évolution concernant les agents intelligents sont les agents mobiles. Ici, les agents peuvent demander des services à d'autres, mais ils ont la possibilité de demander à un agent de s'exécuter sur la même machine que lui, puis de disparaître lorsque sa tache est effectuée. Un agent peut donc voir son exécution interrompue, être déplacé vers une autre machine et son exécution reprendre de manière transparente.
Il est intéressant de voir comment sont gérée cette mobilité et le déplacement des agents sur un réseau.

4.1 Fonctionnement

Pour que le déplacement d'un agent sur le réseau soit possible, il faut que le code de l'agent soit déplacé également. Lors de ce déplacement, il faut que les données envoyées soient reconnues comme programme agent et non comme simples données. Afin d'assurer cette reconnaissance des programmes d'agent, il faut qu'un programme se déroulant en tache de fond (un démon) appelé serveur d'agent.

De plus, ce serveur d'agent doit pouvoir exécuter le code de ces agents indépendamment de la plate-forme utilisée par la machine. Ce problème est résolu par la mise en place d'une machine virtuelle qui, comme la WAM ou la machine java, fourni une indépendance vis à vis du système d'exploitation utilisé et des spécification techniques de la machine.

La figure suivante illustre les différents composants dont je viens de parler.

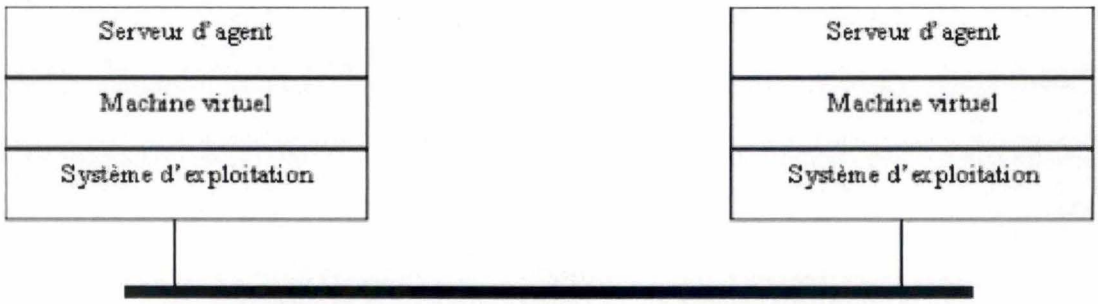

Figure 16 - fonctionnement des agents mobiles [agent2]

Enfin, il ne manque plus qu'un protocole définissant le dialogue entre serveurs d'agent pour gérer la transmission des programmes agents. Ici le protocole ATP sera présenté.

4.2 ATP

Ce protocole, du niveau applicatif sur le modèle OSI, est basé lui aussi sur l'échange de messages de type question-réponse.

ATP se base sur quatre types de messages de requête standard :

- Dispatch : Requête de dépêche : accepter mon agent.
  Un service d'agents A établit une connexion avec un service d'agents B afin de lui demander d'accepter un agent. Sil le service B accepte, il reconstruit l'agent contenu dans le corps du message et redémarre son exécution dans ce nouveau contexte. Finalement il répond à A en fournissant un code de statut.

- Retract : requête de retrait : renvoyez-moi mon agent.
  Un service d'agent A demande à un service d'agent B de stopper l'exécution de son agent et de le lui restituer. B répond en fournissant un code de statut et l'agent spécifié dans le corps de sa réponse.

- Fetch : requête de rapatriement : renvoyez-moi la classe X.
  Pour s'exécuter, un agent peut avoir besoin de certaines ressources qu'il demande à un service d'agent distant. La réponse contient un code de statut et le code exécutable de la ressource exigée. Dans le cas du paradigme orienté objet, la réponse sera une classe.

- Message : requête de message : toute direction.
  Permet aux agents de communiquer avec leurs environnements.

Chapitre 6 : Applications concrètes

1. Pillow

La première application concrète que je vais présenter, consiste plutôt en une librairie de prédicats Prolog orientés Web, qui permet d'effectuer des opérations d'écriture et de manipulation de contenu web.

Cette librairie a été créée par le groupe CLIP de l'université polytechnique de Madrid. Parmi les membres de ce groupe de recherche on peut citer Manuel Hermenegildo et Daniel Cabeza, qui en sont les principaux protagonistes.

Pillow a été développée pour faire partie intégrante du Prolog Ciao mis au point par le même groupe de recherche. Cependant elle a été adaptée à de nombreux autres interpréteurs du langage Prolog tels que Sictus Prolog par exemple.

Au vu du peu de documentation traitant de son implémentation, je vais présenter cette librairie de façon assez pratique, sous la forme d'un didacticiel visant à créer des pages web contenant des formulaires et des script CGI manipulant ces formulaires ; tout ceci au travers de la librairie Pillow. L'étude de cette librairie me semble intéressante dans le cadre de ce mémoire de par le fait qu'elle peut servir de base de communication pour la création de la librairie qui fait l'objet de cet article.

1.1 Création d'un script CGI basic

Comme il a déjà été présenté dans le chapitre 2 consacré aux CGI, un script CGI est une application externe au serveur http, qui est appelée par lui via l'interface CGI.
L'exemple qui suit est un simple script CGI écrit en Prolog et qui ne nécessite pas encore le recours à la librairie :

Exemple 15 - script cgi avec Ciao Prolog

```
main :-
        write ('content-type: text/html'), nl, nl,
        write('<html>'),
        write('hello world'),
        write('</html>').
```

Il s'agit ici, simplement d'écrire une page web sur laquelle est écrit le très célèbre « hello world ». Ce code doit être maintenant compilé avec le CIAO par cette instruction : *« ciaoc –o hello_world.cgi hello_world ».*
L'exécutable qui en résulte doit être placé dans un répertoire accessible par le serveur web et avoir les droits nécessaires pour être exécuté par celui-ci.

1.2 Manipulation de formulaires

Le script précédent produit un résultat qui n'est pas une fonction des paramètres d'entrées qu'il reçoit, ce qui représente un intérêt tout limité. Pour devenir beaucoup plus intéressants, les scripts CGI doivent pouvoir recevoir des données en étant combinés à des formulaires de pages internet. Ces formulaires sont des parties de documents html, qui comportent des champs spéciaux (texte, boutons radio, boite à cocher ...), qui peuvent servir de valeur d'entrée à des scripts CGI.

Dans le cas d'un formulaire, lorsque celui-ci est envoyé au serveur web, les informations contenues dans ce formulaire sont transmises au script CGI qui manipule ces données, via l'entrée standard ou l'URL (cfr. Chapitre 2).

Voici maintenant un exemple de script CGI manipulant les données d'un formulaire, écrit en Prolog à l'aide de la librairie Pillow.

Exemple 16 - gestionnaire de formulaire avec Pillow

```
:- include(library(pillow)).

Main(_) :-
        get_form_input(Input),
        get_form_value(Input, person_name, Name),
        write('content-type : text/html'), nl, nl,
        write('<html><title>numéros de téléphone</title>'),nl,
        write('<h2>numéros de téléphone</h2><hr>'),
        write_info(Name),
        write('</html>').

write_info(Name) :-
        form_empty_value(Name) -> write('Donnez un nom') ;
        Phone(Name, Phone -> write('le numéro de '), write (Name), write(' est le '), write(Phone) ;
        write('Pas de numéro pour '), write (Name).

phone(daniel, '123-456').
phone(manuel, '456-123').
phone(bruno, '987-6547').
```

Dans l'exemple 14, il est à noter les prédicats fournis par la librairie et qui masquent le protocole de bas niveau qui est derrière la gestion de l'interface CGI.

get_form_input(Dic) : traduit l'entrée du formulaire en un dictionnaire qui consiste en des tuples « attribut= valeur ». La valeur des champs vides est traduite par un atome '$empty'.

get_form_value(Dic, Var, Val) : récupère la valeur de la variable Var dans le Dictionnaire Dic.

form_empty_value(Var) : permet de vérifier si la valeur de Var (une zone de text) est vide.

form_request_method(Method) : récupère dans Method la méthode d'invocation du formulaire (Get ou Post).

Ce code assez simple, mais les appels à *write* contenant des marqueurs html ne le rendent pas très lisible. De plus, il n'existe pas de séparation entre le traitement et la gestion des entrées/sorties, alors que ce serait préférable. Pillow fournit des prédicats permettant de traduire le code html en tant que termes Prolog afin de les manipuler plus aisément.

1.3 Traitement du HTML sous forme de termes Prolog

Les structures html ne doivent être traduites en code html qu'au moment même de leur envoi vers la sortie standard en vu de leur affichage. Le reste du temps dans le programme Prolog, les structures html seront traitées sous forme de termes. Voici maintenant les prédicats qui permettent de traduire ces structures en termes.

output_html(F) : prend dans F, un terme html (ou une liste de termes) et envoi à la sortie standard le texte résultant de la traduction des termes dans le format html.

html2terms(Chars, Terms) : met en relation une liste de termes html et une liste de caractères ascii correspondant au format html.

Il est à noter que les termes html ont une structure récursive dans le sens où un terme html peut être à son tour une liste de termes html, dans le cas des balises html contenant des arguments.

Il existe des prédicats de la librairie Pillow qui permettent de traiter les termes html pour représenter les structures html. Je vais présenter celles de bases, en laissant le loisir aux personnes intéressées de consulter le didacticiel de la librairie Pillow `[pillow2]`.

Avant de citer ces quelques prédicats, je tiens à apporter une dernière précision. Il faut distinguer deux types de structure en html :

- les éléments html : sous la forme *'<NAME Attributes>'* ou NAME est le nom de la balise html et Attributes une liste (qui peut être vide) d'arguments pour cette balise.
- les environnements html : sous la forme *'<NAME Attributes> Text </NAME>'* ou NAME définit le nom de l'environnement et Attributes est semblable aux éléments html.

Pour représenter ces deux types de construction, Pillow fournit ces structures Prolog :

- *name $Atts* : Qui représente un élément html de nom 'name' et possédant une liste d'arguments Atts.
  Voici un d'utilisation de cette structure et de sa traduction en html
  'img$[src='images/map.gif', alt= "A map", ismap]' qui devient <img src="images/map.gif" alt="A map" ismap>

- *name(Text)* : qui représente un environnement html de nom 'name' et contient le texte Text.
  Par exemple : address('jlefevre@info.fundp.ac.be') qui devient
  <address> jlefevre@info.fundp.ac.be </address>

- *name(Atts, Text)* : équivalent à la structure précédente, mais ajoutant des attributs à l'environnement html.

La gestion des formulaires se fait également à travers des prédicats fournit par la librairie.

- *start_form(Addr[,Atts])* : définit le début d'un formulaire avec l'adresse du programme qui gérera le formulaire et les attributs facultatifs. En l'absence d'attributs, la méthode par défaut est POST.
  Il existe également start_form qui n'attribue pas d'adresse et prend donc l'adresse du cgi qui génère le script.
- *end_form* : définit la fin du formulaire.
- *input(Type, Atts)* : définit une de saisie d'un certain Type avec une liste d'attributs.

Il existe d'autres prédicats créant d'autres types de zone de saisie de données pour les formulaires; leur structure étant semblable, je ne les présente pas. Elles sont également présentes dans le didacticiel de la librairie Pillow.

1.4 Utilisation des modèles de documents

Maintenant que j'ai présenté comment gérer un formulaire, éditer du html sous forme de termes, on peut remarquer que la présentation des données et leur traitement sont toujours réalisés ensemble. La présentation de la page web étant décrite dans le fichier Prolog, il faut modifier celui-ci (donc le recompiler) pour modifier la présentation du contenu.

Une solution existe : l'utilisation de modèle de document (templates).

Il s'agit d'externaliser la présentation de la page internet dans un fichier html classique. D'ajouter à celui-ci des marqueurs que le programme Prolog pourra trouver afin d'y insérer ses données. Une fois que le fichier modèle est lu par Pillow, ces marqueurs sont vus comme des variables libres sous forme de termes. ces variables seront alors garnies par le programme Prolog avant d'être envoyé vers la sortie standard.

Pour traiter ces modèles de documents, il faut introduire de nouveaux prédicats :

- *html_template(Chars, Terms, Dict)* : parse la chaîne de caractère Chars en tant que modèle html et unifie Terms avec la liste des termes html marqués dans le fichier modèle. Ce marquage se fait à l'aide de l'environnement html *'<V>nom de variable</V>'*.

- *file_to_string(fichier, Chars)* : converti un fichier en une liste de code de caractère. Ce prédicat fait partie du Ciao prolog et non pas de la librairie Pillow.

Pour illustrer tout ceci, voici un exemple complet utilisant un formulaire et une modèle html. Le programme Prolog est à la fois le gestionnaire de formulaire et le générateur de ce formulaire (issus de `[pillow2]`).

Exemple 17 - modèle de document html (fichier Prolog)

```
:- include(library(pillow)).
:- use_module(library(file_utils)).

main(_) :-
        get_form_input(Input),
```

```
        get_form_value(Input, person_name, Name),
        response(Name, Response),
        file_to_string('TlfDB.html', Contents),
        html_template(Contents, HTML_terms, [response = Response]),
        output_html([cgi_reply[HTML_terms]).

response(Name, Response) :-
        form_empty_value(Name) -> Response = [];
        phone(Name, Phone) -> Response = ['Telephone number of ', b(Name), ': ', Phone, $];
        Response = ['No telephone number available for ', b(Name), '.',$].

phone (daniel, '123456').
phone (manuel, '456789'),
phone (macha, '789123').
```

Exemple 15 - modèle de document html (fichier html)

```
<html><head><title>Telephone database</title></head>
<body bgcolor=grey><h2> Telephone database</h2><hr>
<V>response</V>
<form method=post>Enter a name of clip member
<input type=text name='person_name' size=20></form>
</body></html>
```

Comme avant, il y a traitement du formulaire, remplissage de Response. Ensuite, le fichier html est transformé en une chaîne de caractères. Cette chaîne est transformée en termes grâce au *html_template* où les zones marquées sont remplies. Enfin, il y la réponse cgi est générée vers la sortie standard.

Maintenant, le traitement et la présentation sont maintenant séparés. Cependant, le CGI bien qu'efficace, n'a qu'une durée de vie limitée. En effet, à chaque interaction le contrôleur du formulaire (le script cgi) est lancé et doit se terminer. Ceci représente deux inconvénients.
D'abord, il n'existe pas de sauvegarde d'état d'une requête à une autre; bien que ce soit possible via des cookies ou un fichier de configuration.
Ensuite, lancer et arrêter régulièrement l'application peut diminuer l'efficacité, surtout si l'application doit consulter une grande base de données.

Une solution apportée par la librairie Pillow est de modifier l'architecture CGI en introduisant la notion de modules actifs.

1.5 Les modules actifs

L'idée principale est illustrée à la **Erreur ! Source du renvoi introuvable.**. L'opération de requête est identique jusqu'au point 3. A cette étape, le contrôleur de formulaire qui est lancé n'est plus l'application principale, mais une interface vers l'application réelle, qui tourne en permanence sur le serveur. Ceci lui permet de maintenir un état continuel de l'application.

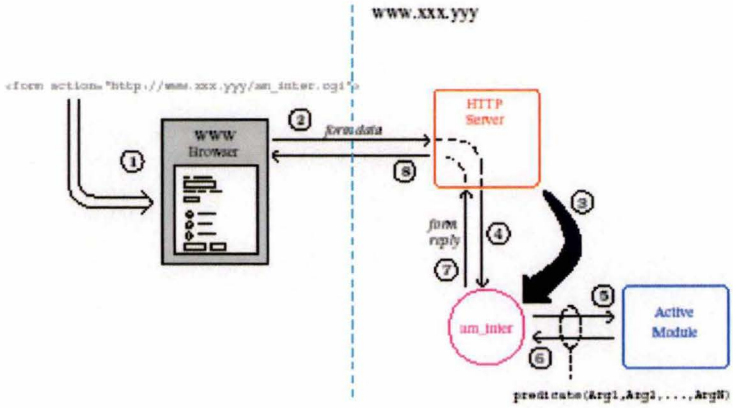

Figure 17 - Utilisation d'un module actif

Il n'y aura que l'interface qui sera lancée et arrêtée à chaque requête. Cette interface transmet les données d'entrée reçues par le serveur (4) vers l'application (5). Ensuite, elle transmettra les résultats vers le serveur http avant de se terminer (6).

La partie intéressante réside dans la communication entre l'interface et l'application.
Un modèle actif est un module ordinaire auquel des ressources de calcul lui sont alloué par le système (un processus en Unix par exemple) et qui est situé sur un socket sur le réseau. La compilation d'un tel module produit un exécutable qui, lors de son exécution, agit en temps que serveur pour un certain nombre de relations – représentant les prédicats exportés par le module.

Ces relations peuvent être accessibles par tout programme sur le réseau, en chargeant le module et donc en chargeant ces relations distantes. Il s'agit en fait, au lieu de charger le code du module, de mettre en place des moyens permettant aux appels locaux d'être exécutés en temps qu'appels à distance vers ce module actif. Enfin, un module actif doit faire connaître son adresse réseau pour pouvoir être utilisé.

Voici maintenant les prédicats fournis par le Ciao Prolog (et non pas Pillow) permettant de gérer ces modules :

- *:- use_active-module(Module, Predicates)* : déclaration permettant d'importer les prédicats dans la liste Predicates à partir du module actif Module. Cette déclaration faite, le code est semblable à l'utilisation de use_module.

- *module_address(Module, Address)* : renvoi dans Address, l'adresse de Module.

- *save_addr_actmod(Address)* : constitue un moyen de publier l'adresse d'un module actif.

- *make_actmod(ModuleFile, PublishModule)* : crée un module actif exécutable à partir du fichier source ModuleFile, en utilisant l'adresse du module de publication. Lors du lancement du module, un socket est créé, et le prédicat précédent est utilisé pour publier l'adresse.

Ce mécanisme de publication est assez flexible car il permet de définir la manière dont les modules actifs sont localisés. Pour ce faire, il suffit d'écrire un couple de librairies : en définissant la manière de publier une adresse et l'autre la manière de localiser cette adresse. Pour exemple, parmi les librairies standard du Ciao Prolog, deux librairies définissent ces mécanismes. Ces dernières utilisent un répertoire accessible à toutes les machines concernées, leur permettant d'écrire et de consulter des adresses de modules actifs. Le prédicat module_address/2, examine ce répertoire pour y trouver les données nécessaires. Une autre solution possible est de reprendre l'idée du naming service utilisée dans la gestion des systèmes distribués (tels que Corba). Il s'agit de poster l'adresse du module vers une adresse internet où un serveur de nom, disposant d'une adresse fixe va traiter ces enregistrements d'adresses.

Si l'on considère un point de vue plus axé sur l'implémentation, les modules actifs sont essentiellement des démons : il s'agit de processus indépendants lancés au niveau du système d'exploitation. Dans le Ciao Prolog, la communication se fait au travers de sockets, ainsi les adresses des modules sont des sockets sur une machine. Les requêtes sont donc envoyées aux travers de ces sockets vers les programmes distants. Le démon reçoit une de ces requêtes, il la capture, la traite et renvoi sur le socket le résultat de son traitement.

Pour terminer cette description des modules actifs, voici le code d'un module actif et d'un gérant de formulaire qui va interroger ce module actif. Le code du module actif doit être compilé comme suit : *make_actmod(phone_db, 'actmods/filebased_publish')*

Exemple 18 - Code source d'un module actif

Voici le code source du module actif(phone_db)

```
:- module(phone_db, [response/2]).

response(Name, Response) :-
                form_empty_value(Name) -> Response = 'Entrez un numéro de téléphone.'
                ; phone(Name, Phone) -> Response = ['Le numéro de ', b(Name), ': ', Phone]
                ; Response = ['Pas de numéro disponible pour ', b(Name), '.'].

:- dynamic phone/2.
phone(daniel, '123456').
phone(manuel,'456789').
phone(bruno, '789456').
```

Voici le code source du gestionnaire de formulaire : il va servir d'interface vers le module actif.

```
#! /usr.local/bin/ciao-shell

:- use_active_module(phone_db, [response/2]).
:- use_module(library('actmods/filebased_locate')).
:- include(library(pillow)).

main(_) :-
        get_form_input(Input),
        get_form_value(Input, person_name, Name),
        response(Name, Response),
        output_html ([
```

```
            cgi_reply,
            start,
            title('Base de données téléphoniques'),
            image('phone.gif'),
            heading(2, 'Base de données téléphoniques'),
            --,
            Response,
            $,
            start_form,
            'Entrez un nom, puis pressez Enter'),
            \\,
            input(text,[name=person_name, size=20]),
end_form,
end]).
```

1.6 Implémentation en Ciao Prolog

Nous allons maintenant observer les interactions qui se produisent avec le moteur du Ciao Prolog lors de l'exécution d'un script cgi réalisé à l'aide de la librairie Pillow.

Nous allons nous intéresser aux deux principales actions d'échange entre le monde web et le Prolog : la consultation des données d'un formulaire à travers l'interface CGI et l'envoi de la réponse vers le client web.

Lors de l'émission d'un formulaire, le serveur web charge les variables d'environnement avec le contenu les informations de la requête. Ensuite, il appelle l'application extérieure. Ici, il s'agit d'un fichier cpx qui correspond à un programme Ciao auto exécutable. Lors de l'appel à celui-ci, le moteur du Ciao vers se lancer et charger le programme. Ce moteur se charge d'initialiser une WAM, c'est à dire d'allouer de la mémoire pour les différentes zones mémoires (HEAP, STACK, PDL, TRAIL, ...) ainsi que les pointeurs vers ces structures et de les initialiser. Ensuite, le code est chargé dans la Code Area et il est interprété. Le moteur à alors accès aux différentes variables d'environnement.

Le moteur du Ciao étant implémenté en C, une petite application a été réalisée de ce langage C, qui illustre l'accès aux différentes variables d'environnements utilisées par l'interface CGI. Une fois les données récupérées, elles sont envoyées au navigateur web pour affichage, via la sortie standard. Le code cette petite application est fourni en annexe 2.

Lors de l'exécution, l'accès aux données d'un formulaire se fait suite à l'instruction get_form_input(Dict). Dans la librairie pillow ce prédicat est implémenté comme suit :

```
get_form_input(Dic) :-
    form_request_method(M),
    get_form_input_method(M, Dic), !.
get_form_input([]).

get_form_input_method('GET', Dic) :-
    ( getenvstr('QUERY_STRING',Q), Q \== [] ->
      append(Q,"&",Cs),
      form_urlencoded_to_dic(Dic, Cs, [])
    ; Dic = [] ), !.
```

```
get_form_input_method('POST', Dic) :-
    getenvstr('CONTENT_TYPE', ContentType),
    http_media_type(Type, Subtype, Params, ContentType,[]),
    get_form_input_of_type(Type, Subtype, Params, Dic), !.

get_form_input_method(M, _) :-
    html_report_error(['Unknown request method ', tt(M),
                ' or bad request.']).

form_request_method(M) :-
    getenvstr('REQUEST_METHOD', MS),
    atom_codes(M,MS).
```

Le prédicat get_input_form, après avoir vérifié la méthode d'invocation (Get ou Post), lance la lecture des données en fonction de la méthode obtenue. Cette lecture se réalise grâce à la procédure get_form_input_method (Method, Dic). Celle-ci fait appel au prédicat getenvstr(Var, Content) qui demande au moteur Ciao de lire la variable d'environnement Var et de l'ajouter dans HEAP et de l'unifier à la variable Content. Une fois le contenu de l'environnement disponible à l'aide de la variable Content, ce contenu peut être parsé pour être présenté sous forme d'une liste ayant la forme [Nom = Valeur].

La seconde interaction concernant l'écriture du résultat sur la sorite standard est mise en œuvre plus simplement.

Le prédicat output_html effectue deux tâches. La première consiste en une simple traduction de termes html vers une chaîne de caractères contenant du texte formaté en html.

```
output_html(F) :-
    html_term(F,T,[]),
    write_string(T).
```

À titre documentaire, voici un des prédicats de la procédure html_term, traduisant l'introduction d'une image dans le code html.

```
html_term(image(Addr, Atts)) --> !,
    "<img",
    html_atts([src=Addr|Atts]),
    ">".
```

La procédure htlm_atts/1 parcoure la liste des attributs et les ajoutes à la chaîne de caractères.

Enfin, write_string(T), demande au moteur Ciao d'envoyer la chaîne de caractères, maintenant formatée en html, vers la sortie standard à fin que cette chaîne de caractère soit affichée par le navigateur web.

Pour réaliser cet envoi vers la sortie standard, le moteur du Ciao, lorsqu'il traitera cette instruction aura chargé le contenu de la variable T dans son registre d'argument, depuis le HEAP.

Il s'agira ensuite d'envoyer le contenu de ce registre vers la sortie standard. Cet envoi se réalise en une instruction C : *printf.*

La présentation de ces deux interactions permet de voire se rendre compte de l'abstraction qui est fournie par la librairie Pillow pour le développeur d'applications Internet.

2. LogiMoo

Cette section consacrée à l'étude de l'application LogiMoo se base essentiellement sur les informations contenues dans l'article [moo1].

LogiMoo est un monde virtuel implémenté en BinProlog. Ce monde virtuel est destiné à tourner dans un navigateur internet (tel Internet Explorer ou Netscape) et qui permet aux utilisateurs de travailler ensembles et de partager des lieux, des objets ou des agents virtuels. Il existe d'autres mondes virtuels rassemblés en deux grandes familles, les MUD's et les MOO's (pour Multi User Domains – Object Oriented) : des domaines pour plusieurs utilisateurs échanger des informations dont certains sont orientés objet.
Implémenté au-dessus de BinProlog, il utilise un langage de coordination du même style que le langage Linda.
Son interface utilisateur est gérée en langage naturel ce qui signifie qu'il est en mesure de comprendre et d'interpréter le langage naturel comme langage de commande (par ex. se mouvoir dans le monde virtuel). De plus, LogiMoo peut compléter son langage naturel par l'apprentissage de nouveaux mots, adverbes et verbes au fur et à mesure de son utilisation.

2.1 Architecture de LogiMoo

Il s'agit d'une architecture en couche. Voici les couches principales qui constituent ce système.

- Le système BinProlog en tâche de fond, ce qui permet en plus de l'interpréteur du langage, de supporter les CGI et la programmation Client/Serveur.
- Les opérations de base LogiMoo, qui fournit des opérations Moo implémentées en tant que prédicats Prolog compilés.
- Le compilateur du langage naturel, qui traduit les phrases en commandes pour LogiMoo.
- L'interface utilisateur : permet de créer des objets persistants sur le serveur ; objets résultats du langage naturel ou des commandes de base BinProlog.

Voici une représentation du déploiement de LogiMoo à travers internet utilisant un navigateur Netscape

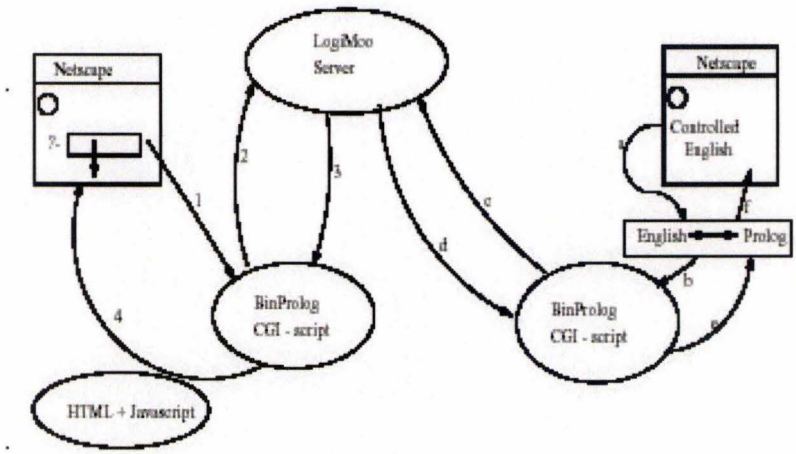

Figure 18 - déploiement d'une application LogiMoo

Une interface CGI en BinProlog entre le navigateur et le serveur distant LogiMoo interagissent. Le CGI va envoyer et recevoir les données du serveur et va pouvoir les afficher en contenu web sur le navigateur.
Si l'utilisateur encode ses requêtes en langage naturel, il faut passer avant par un traducteur vers les prédicats LogiMoo.
Pour le serveur LogiMoo, les objets sont représentés sous forme d'URLs vers les pages web de leur propriétaire. Leur représentation réelle se trouve du coté client chez ce propriétaire. LogiMoo est donc considéré comme un gestionnaire d'objet de haut niveau.

Les activités de base du serveur LogiMoo permettent de :

- Créer et placer des objets.
- Les déplacer d'un endroit à un autre.
- Les donner ou les vendre.
- Dialoguer (en privé ou en public).

2.2 La coordination

La coordination utilisée pour assurer le déploiement du système est de type Linda, utilisant donc une mémoire partagée.

Pour fonctionner, Bin Prolog utilise un espace de tuples semblable à celui utilisé par Linda.
Les premières propriétés de ce système sont d'assurer la persistance des données, leur manipulation associative et un accès synchroniser leurs accès. La synchronisation des données assure leur cohérence et leur intégrité puisque le serveur sera utilisé par plusieurs utilisateurs en parallèle.

Le système utilise un sous-ensemble d'opérations très proche du Linda original que j'ai déjà présenté. Voici les primitives principales de LogiMoo pour gérer une mémoire partagée.

Out(X) insérer X sur le serveur dans la mémoire partagée.

In (X)	le processus attend de pouvoir trouver un objet correspondant à X sur le serveur et le prend
All (X ,Xs)	renvoi Xs (la liste des objets correspondant à X sur le serveur)
Run (Goal)	lance un thread qui exécute Goal
Local_out (X)	insère un objet privé X sur le « tableau » local.
Local_rd (X)	vérifie s'il existe un objet correspondant à X sur la « boit noire » locale.

Il existe encore quelques opérations construites au-dessus des primitives LogiMoo.

rd (X)	vérifie si un objet correspondant à X est sur le serveur.
cout (X)	insère X sur le serveur sauf s'il existe déjà un objet y correspondant.
cin (X)	prend un objet correspondant à X du serveur et échoue s'il n'y en a pas.
forall (X)	exécute G pour tous les objets correspondant à X.

Voici un exemple d'utilisation de la coordination Linda de LogiMoo

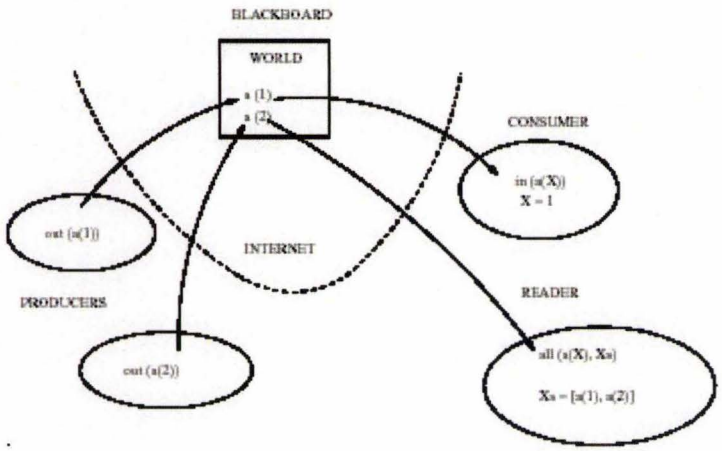

Figure 19 - coordination Linda dans LogiMoo

Le premier producteur introduit le prédicat a(1) en mémoire partagée.
Le second producteur introduit le prédicat a(2) en mémoire partagée.
Le deuxième consommateur demande la liste des prédicats correspondants à a(X). il reçoit la liste Xs = [a(1), a(2)].
Le premier consommateur demande un prédicat pouvant être mis en correspondance avec a(X). Il reçoit en retour la valeur avec laquelle son inconnue, sa variable, a pu être mise en correspondance X=1. L'objet demandé est retiré de la mémoire partagée.

2.3 Le noyau de LogiMoo

Les actions en LogiMoo sont un ensemble de prédicats Prolog qui masquent la complexité de la communication distribuée (ports, possessions des objets,...).
LogiMoo doit donc être vu comme un tableau qui permet partager et d'agir sur des objets.

Les informations non partagées restent du coté du client donc sa boite noire locale. On y retrouve ainsi son nom. Pour le retrouver il existe le prédicat whoami (X) :- local_rd(iam(X)). Il existe

également un prédicat qui permet de retrouver où un utilisateur (son avatar) se situe : Whereami(P) qui unifie P avec la place où se situe l'avatar.
Cette utilisation de la boite noir en local permet d'une part d'accéder de manière plus rapide aux données privées et d'autre part de mettre en œuvre des protocoles de sécurité.

Entrons maintenant un peu plus profondément dans ce tableau et nous intéressons aux ports de communication entre les différentes places. Ces places sont des objets que l'on peut créer et manipuler dans le « monde » virtuel de LogiMoo. Les places peuvent ouvrir des ports pour communiquer entre elles en échangeant des objets par ces ports.

Voici les primitives Prolog qui sont derrière ses actions :
dig(Place)
port(P1,Dir,P2)
move (O,P1,P2) :- cin(contains(P1,O)), out(contains(P2,O)).

Pour déplacer un objet entre deux places, il regarde dans la mémoire partagée si la place contient cet objet et s'il y est, il le retire. Ensuite il ajoute que la place 2 contient cet objet.

Au-dessus du déplacement d'objet on trouve encore les primitives permettant de déplacer un utilisateur (go), d'avoir la possession des objets (craft) et de donner ses objets à un autre utilisateur (give).

Voici ces prédicats :

```
go(Dir) :-
        whereami (Place),
        rd (port (Place, Dir, NewPlace)),
        whoami (Me),
        move (Me, Place, NewPlace),
        forall(has(Me, O), move(O, Place, NewPlace)).
```

Pour déplacer un utilisateur on utilise whereami pour mettre dans Place son emplacement, avec rd, on vérifie qu'il existe un port vers la destination et on récupère la nouvelle place en fonction de la direction. Ensuite on récupère le nom de l'utilisateur et on déplace cet utilisateur de son ancienne place vers la nouvelle grâce au move. Enfin le forall prends tout les objets que l'utilisateur possède (« has ») et les déplace également.

```
craft (O) :-
        whoami( Me),
        rd (contains(Place, Me)),
        out (contains(Place, O)),
        out (has (Me, O)).
```

Pour prendre possession d'un objet, on récupère d'abord le nom de l'utilisateur avec whoami, on regarde dans quelle place on se trouve. Enfin on ajoute que l'objet est au même endroit et que l'utilisateur le possède (« has »).

```
gives (From, To, O) :-
        cin (has (From, O)),
        out (has (To, O)).
```

Pour donner un objet on utilise le retrait conditionnel (« cin ») pour vérifier si l'émetteur possède l'objet et retirer cette information de possession du monde virtuel. Enfin on ajoute que l'utilisateur cible possède cet objet.

```
give (Who, What) :-
        whoami (Me),
        cin (has (Me, What)),
        out (has (Who, What)).
```

Ici l'on procède de manière similaire, la seule différence est que l'on considère l'émetteur comme l'utilisateur courant. Le nom de cet utilisateur sera récupéré grâce à whoami.

2.4 Déploiement de LogiMoo

LogiMoo est déployé en tant qu'application web de type client/serveur. LogiMoo permet de s'interfacer avec différentes plates-forme, je ne vais ici présenter que l'interfaçage avec l'internet car c'est celui dont il est question dans ce présent mémoire.

Concernant cet interfaçage internet, il utilise certaines fonctionnalités avancées des navigateurs web. On y compte les frames et formulaires gérés par internet explorer ou netscape, le javascript pour que bin prolog puisse contrôler les frames netscape (ou navigateurs compatibles), la gestion des script CGI de Bin Prolog et le plugin WorldView pour la navigation VRML[2].

La représentation des objets dans le système se fait au moyen de lien URL's vers la page du propriétaire où ils sont représentés en différents formats (HTML, GIF, JPG, VRML,...). Ces liens sont gardés par le serveur LogiMoo et les objets cibles sont sur les ordinateurs des utilisateurs ; ceci lui permet donc de mettre à jour l'objet sans devoir en avertir les serveurs.
De plus, sous netscape l'état local n'est pas sauvegardé. Pour palier à ce petit défaut, le CGI de BinProlog utilise les formulaires dont l'action de soumission provoque une connexion avec un serveur LogiMoo persistant. C'est ce serveur qui gardera les états des différents utilisateurs connectés. Cette sauvegarde d'états se fait via un thread (pour pouvoir continuer de fournir le service aux utilisateurs pendant la sauvegarde) qui enregistre les états dans un fichier.

En ce qui concerne les requêtes, elles sont transmises au serveur LogiMoo via une interface CGI de type Post (par l'entrée standard). Le serveur lit cette entrée à l'aide de la variable de longueur (content_length). Ensuite les espaces superflus sont retirés et la requête avec ses variables est extraite via une liste de conversion en termes. Enfin, il y a identification auprès du serveur et lancement de la requête. Voici comment est faite cette transformation.

[2] VRML : Virtual Reality Modeling Language (langage de modélisation de réalité virtuelle) qui utilise la 3D pour représenter un monde virtuel et interagir avec celui-ci. Mis au point par Silicon Graphics. Pour plus d'informations consultez le site http://apia.u-strasbg.fr/vrml/

```
:- op((1199, xfy, (&)).

(login =L & passwd=P & home=H & query=Query) :-
        login(L,P,H),
        metacall(Query).
```

Avec LogiMoo et sa représentation à l'aide de tableaux pour représenter les mondes virtuels, il est possible de construire un réseau de MOO's via internet comme moyen de diffusion. Il faut alors distinguer deux types de tableaux : ceux qui tournent sur la machine même et qui fournissent de la synchronisation et de la communication entre différents processus sur cette machine ; et les tableaux virtuels qui représentent l'instance d'exécution d'un tableau à distance. Ces derniers ne contiennent pas de données propres il s'agit d'une représentation du tableau distant, d'un alias. Ils sont utilisés pour fournir une communication avec le serveur, et les opérations effectuées sur un tableau virtuel sont immédiatement reportées vers la boite à distance à l'aide du réseau. Il est possible de chaîner plusieurs tableaux virtuels, le tableau à distance pouvant lui-même est un tableau virtuel. Il faudra tout de même que le dernier tableau à distance soit un tableau physique contenant des données réelles.

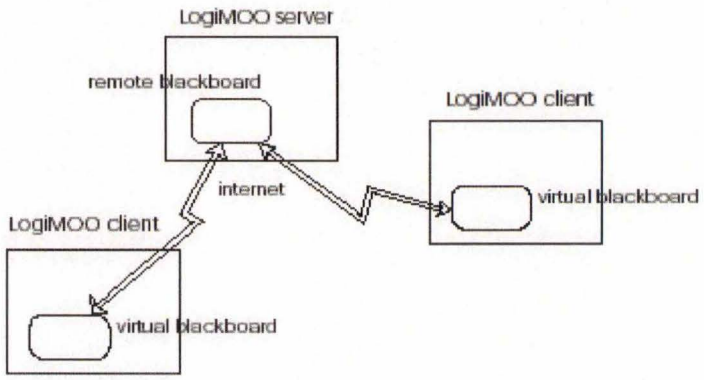

Figure 20 - réseau de MOO's à travers internet

Pour supporter ces concepts, un langage qui se baserait uniquement sur Linda pour gérer la coordination ne conviendra pas car une application Linda est prévue pour gérer un certain nombre de processus lancés par une seule application qui veut résoudre un problème donné. Ici les processus clients peuvent se connecter et se déconnecter à tout moment.
C'est le concept de tableaux virtuels qui permet de combler cette lacune.
Lorsqu'un client désire se connecter à un monde virtuel (MOO), il crée une boite virtuelle locale, se connecte à un MOO de niveau supérieur et dès lors il peut interagir avec le monde virtuel sans contrainte. Linda communiquera avec le tableau virtuel en local, ce que ce langage sait très bien réaliser.

Partie III : Conclusions

Au commencement, il existait deux mondes distincts l'un de l'autre : Internet et la programmation logique. Chacun de ces mondes possède une grande force : Internet constitue un large moyen de diffusion de l'information aux divers format de données et Prolog permettant de décrire de vastes systèmes experts qui visent à apporter ses conseils à qui en a besoin.

L'étude du langage Prolog et des applications issues de la programmation logique (Langage Naturel, Systèmes multi-agents) met en évidence l'intérêt que peut avoir ce monde de l'intelligence artificielle pour un grand nombre d'utilisateurs.

Une collaboration entre ces deux mondes à pu voire le jour via l'interface CGI. Il est alors possible de créer une application Prolog qui sera appelée par un navigateur Web. Enfin, ils peuvent communiquer entre eux ! Cependant, cette communication reste assez complexe dans sa mise en œuvre, pour un vaste public. La librairie Pillow apporte une abstraction supplémentaire dans la manipulation des données provenant des pages web et permet d'atteindre le cercle des développeurs web en leur apportant la puissance du Prolog.

Enfin, la coordination apportée par les tableaux partagés de style Linda, permet de créer des zones de partages de données. Une application frappante de ce partage est l'application LogiMoo qui gère un monde virtuel au travers d'Internet et permet l'échange d'objets virtuels au travers d'un monde commun basés sur ces zones partagées.

Cependant, à mon sens, il ne faut pas s'arrêter en si bon chemin. Certes, il est possible de créer assez facilement des applications logiques distribuées à travers Internet. Mais ces techniques demandent tout de même un certain apprentissage. Je pense que le but ultime, serait de voir apparaître des éditeurs pour de telles applications. Ceux-ci gérant aussi bien la génération de pages internet (html, xml, php), que la conception de bases de connaissances et de systèmes experts.

Bibliographie

[agent1] Le site de l'information décisionnelle, « Les agents intelligents » http://www.decisionnel.net/agentintelligent/, 2002.

[agent 2] St. Angrelot, G. Bonnet, G. Regnault , *« les agents intelligents sur internet »*, Ecole polytechnique de l'université de Nantes, 2000.

[agent3] François Bourdon, *« Cours Systèmes Multi-Agents »*, Université de Caen, 2003.

[bin1] Paul Tarau, *"The BinProlog Experience : Implementing a High_Performance Continuation Passing Prolog Engine"*, University of North Texas.

[bin2] Paul Tarau et Veronica Dahl, *"Logic Programming and Logic Grammars with Binarization and First_order Continuations"*, BinnetCorp.

[bin3] Paul Tarau, "a compiler and a simplified abstract machine for the execution of binary metaprograms", 8° conférence internationale sur la programmation logique, 1991.

[bin4] P. Tarau and M. Boyer, *"Elementary Logic Programs"*, Proceedings of Programming Language Implementation and Logic Programming, numéro 456 pages 159—173, août 1990.

[cgi1] Rotule, *Mini Guides « Les CGI »*, http://guides.rotule.net/cgi/, Rotule 1999 – 2002.

[linda1] Peter Kropf, *«Notes de cours Introduction à Linda»*, www.iro.umontreal.ca/~kropf/ift-6052/notes/linda.pdf, Automne 2000.

[moo1] P. Tarau, K. De Bosschere, V. Dahl, S. Rochefort , *« LogiMoo : an extensible Multi-User Virtual World with Natural Language Control »*, The journal of logic programming, Elsevier Science Publishing, 1993.

[pilow1] D. Cabeza, M. Hermenegildo, *« The pillow web programming library, reference manual»*, The Computational logic, Languages, Implementation, and Parallelism Laboratory, 2001

[prol1] Pierre Bonzon, *« Notes de cours : Informatique théorique »*, Université de Lausanne, 2003.

[vrml] *« Informations générales sur VRML »*, http://apia.u-strasbg.fr/vrml/

[wam1] Hassan Aït-Kaci, *« Warren's abstract Machine, a tutorial reconstruction »*, http://www.isg.sfu.ca/~hak, Février 1999, MIT Press.

[wam2] Jacques Noyé, *« Elagage de contexte, retour arrière superficiel, modifications réversibles et autres : une étude approfondie de la WAM »*, Université de Rennes I, novembre 1994.

[wam3] F. Fages et E. Clergerie , *« Programmation Logique par Contraintes, Projet : Compilateur Prolog en Prolog »*, Avril 1997.

[wam4] Daniel Diaz, *« Etude de la compilation des langages de programmation logique par contraintes sur les domaines finis »*, Université d'Orléans, chapitre 3 , janvier 1995.

[wam5] Emmanuel Atinault Emmanuel. « Cours de Prolog », EFREI, janvier 2002.

Annexes

Annexe 1 : Instructions de la WAM

Put instructions
- put variable Xn_ Ai
- put variable Yn_ Ai
- put value Vn_ Ai
- put unsafe value Yn_ Ai
- put structure f_ Ai
- put list Ai
- put constant c_ A

Get instructions
- get variable Vn_ Ai
- get value Vn_ Ai
- get structure f_ Ai
- get list Ai
- get constant c_ Ai

Set instructions
- set variable Vn
- set value Vn
- set local value Vn
- set constant c
- set void n

Unify instructions
- unify variable Vn
- unify value Vn
- unify local value Vn
- unify constant c
- unify void n

Control instructions
- allocate
- deallocate
- call P_N
- execute P
- proceed

Choice instructions
- try me else L
- retry me else L
- trust me
- try L
- retry L
- trust L

Indexing instructions
- switch on term V_C_L_ S
- switch on constant N_ T
- switch on structure N_ T

Cut instructions
- neck cut
- get level Yn
- cut Yn

Annexe 2 : Application manipulant les variables d'environnements du CGI.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* structure représentant une variable avec son nom et sa valeur
/* cette structure est destinée à former une liste chaînée des variables du formulaire
typedef struct variable{
 char nom[100];
 char valeur[100];
 struct variable* ptrSvt;
} variable;

/*************************
/     PROTOTYPES      *
/*************************/
int contentLength(char* env[]);
variable* queryString(char* env[], variable* vars);
variable* parsePost(char post[], variable* vars);

/*********************************
/     Fonction principale      *
/*********************************/
int main(int argc, char* argv[], char* env[])
{
 char ligne[255];
 int ln_cont = 0;
 variable* tabVar=NULL;
 variable* ptrVar;

 ln_cont += contentLength(env)+1;
 if(ln_cont>1){
  fgets(ligne, ln_cont, stdin);
        tabVar=parsePost(ligne, tabVar);
 }
 tabVar=queryString(env, tabVar);
 printf("content-type: text/html\n\r\n\r");
 printf("<html><h2>Ligne reçue par méthode post : </h2><br>\n\r\n\r");

 ptrVar=tabVar;
 while(ptrVar!=NULL) {
        printf("<b>%s</b> : %s<br>",ptrVar->nom, ptrVar->valeur);
   ptrVar=ptrVar->ptrSvt;
```

```
 }

 printf("</html>");
 return 0;
}

/*********************************************************************
/*  Récupère la longueur des données envoyées par la méthode post  *
/*  @param : char* env[] "variables d'environnements"              *
/*  @out : nombre de byte de données                               *
/*********************************************************************/
int contentLength(char* env[]) {
 int i, ln=0;
 char temp[100];
 char lnTemp[10];
 for(i=0;env[i];i++) {
        strcpy(temp, env[i]);
        if(strstr(temp,"CONTENT_LENGTH")!=0){
                int j=0,k=0;
                while(temp[j++] != '=');
                while(lnTemp[k++] = temp[j++]);
                ln = atoi(lnTemp);
                break;
        }
 }
 return ln;
}

variable* queryString(char* env[], variable* vars) {
 int i, ln=0;
 char temp[1036];
 char lnTemp[1024];
 for(i=0;env[i];i++) {
        strcpy(temp, env[i]);
        if(strstr(temp,"QUERY_STRING")!=0){
                int j=0,k=0;
                while(temp[j++] != '=');
                while(lnTemp[k++] = temp[j++]);
                break;
        }
 }
 return parsePost(lnTemp, vars);
}


variable* parsePost(char post[], variable* vars) {
 char* nom = (char*)malloc(100*sizeof(char));
```

```
char* valeur = (char*)malloc(100*sizeof(char));
char* mark;
char* ptrnom;
char* ptrval;
variable* ptrVar = vars;

if(ptrVar!=NULL)while(ptrVar->ptrSvt!=NULL)ptrVar = ptrVar->ptrSvt;
mark = post;
while(*mark!='\0') {
        ptrnom = nom;
        ptrval = valeur;
        while(*mark!='=' && *mark!='\0')*ptrnom++=*mark++;
        mark++;*ptrnom='\0';
        while(*mark!='&' && *mark!='\0')*ptrval++=*mark++;
        if(*mark!='\0')mark++;
        *ptrval='\0';

        if(ptrVar == NULL) {
                ptrVar = (variable*) malloc(sizeof(variable));
                vars = ptrVar;
        } else {
                variable* ptrNew =(variable*) malloc(sizeof(variable));
                ptrVar->ptrSvt = ptrNew;
                ptrVar = ptrNew;
        }
        ptrVar->ptrSvt = NULL;
        strcpy(ptrVar->nom, nom);
        strcpy(ptrVar->valeur, valeur);
}
return vars;
}
```